Curso técnico em Informática
Soeducar - PEP

Algoritmo e Programação

Alterada por: Daniel Pereira Ribeiro

Sumário

Introdução

Nesta apostila estudaremos **Lógica de Programação** e, para isso, é importante ter uma visão geral do processo de desenvolvimento de programas (software), visto que o objetivo final é ter um bom embasamento teórico para a prática da programação de computadores.

1. Alguns Conceitos Básicos

Dados –São elementos brutos que podem ser processados por um computador para se obter alguma conclusão ou resultado, ou seja, uma informação.

Computador –Máquina (Hardware) muito rápida que pode processar dados, realizando cálculos e operações repetitivas, se dotados de programação adequada (software ou firmware), e que fornece resultados corretos e precisos.

Informação–É o resultado do processamento dos dados pelo computador. Uma informação pode ser considerada como um dado para outro processamento e, por isso, muitas vezes é referenciada como dado de saída.

Hardware–É um termo de origem americana que engloba todo o equipamento, principal e periférico, de um computador. O termo é utilizado também para equipamentos sem software. O hardware normalmente é fixo e difícil de ser modificado.

Software–É outro termo de origem americana que engloba programas, documentação, processamento de dados, utilizados em um computador para resolução de determinado problema. O software, ao contrário do hardware, é maleável e por isso mais facilmente modificável pelo programador, para adaptação a novos dados, novos requisitos do problema a ser resolvido, ou novo hardware onde vai funcionar (ser executado).

Programa de computador –Sequência de instruções não ambígua e finita em uma linguagem de programação especifica que, quando executada, resolve um problema determinado.

Linguagem de máquina –É a linguagem binária (composta de zeros e uns) utilizada pelos computadores, para representar dados, programas e informações. É tediosa, difícil de compreender e fácil de gerar erros na programação.

Linguagens baixo nível –Linguagem de programação que compreende as características da arquitetura do computador. Assim, utiliza somente instruções do processador, para isso é necessário conhecer os registradores da máquina. Nesse sentido, as linguagens de baixo nível estão diretamente relacionadas com a arquitetura do computador. Um exemplo é a linguagem Assembly que trabalha diretamente com os registradores do processador, manipulando dados.

Linguagens alto nível –Englobam todas as linguagens de programação que utilizam compiladores ou interpretadores. Possuem instruções mais poderosas que as linguagens de baixo nível, facilitando ainda mais o trabalho do programador.
Ex. Pascal, Delphi, C, C#, Java, PHP, etc.

Linguagens não procedurais de alto nível –São linguagens utilizadas para sistemas de gerenciamento de banco de dados, planilhas, e outros aplicativos que utilizam comandos ainda mais poderosos que as linguagens de alto nível e, por isso, são ainda mais fáceis de programar.
Ex. SQL, DBASE, Oracle PL/SQL, XML, etc.

Compilador –É um programa utilizado para traduzir os programas escritos pelo programador nas linguagens de alto nível (programa fonte) para linguagem de máquina (programa executável), para que possa ser executado pelo computador.

Interpretador –É um programa que traduz os programas escritos pelo programador para linguagem de máquina no momento da execução (não existindo assim o programa executável).

IDE –(Ambiente de Desenvolvimento Integrado) É um programa que reuni algumas ferramentas para o desenvolvimento e software como, editor de texto, compilador, depurador, entre outro, facilitando o trabalho do programador.

2. Software

O software de um computador é o que determina o seu uso e os resultados que serão produzidos e apresentados. Em um computador digital existem diversos tipos diferentes de software com finalidades e complexidades diferentes. Normalmente, quanto mais relacionado e próximo ao hardware, o software é mais difícil e complexo de ser desenvolvido e mantido pelo programador. A Figura abaixo procura ilustrar em camadas o que foi dito. Partindo-se do hardware que está no centro e normalmente contém um conjunto de instruções operacionais programado diretamente nele (FIRMWARE).

O firmware é armazenado permanentemente num circuito integrado (chip) de memória do hardware, como uma ROM, PROM, EPROM ou ainda EEPROM e memória flash, no momento da fabricação do componente. Muitos aparelhos simples possuem firmware, entre eles podemos citar controle remoto, calculadora de mão, alguns periféricos do computador como disco rígido, teclado, modem e também aparelhos mais complexos como celulares, câmeras digitais, aparelhos de som, geladeiras, entre outros que possuem um firmware para a execução de suas tarefas.

O Sistema Operacional com seus diversos programas é a primeira camada de software do nosso computador. É muito complexo desenvolver-se um sistema operacional como DOS, UNIX, LINUX, WINDOWS, MAC OS, ANDROID e outros que você já deve ter ouvido falar.

Externamente ao sistema operacional vamos encontrar os compiladores e interpretadores das linguagens de programação que, de certa forma, traduzem os programas aplicativos para a linguagem de máquina que o computador entende. Nesta segunda camada encontramos também os processadores ou editores de texto, os gerenciadores de bancos de dados (MySQL, Firebird, PostgreSQL, Oracle, MS Access, etc.), as planilhas e muitos outros programas utilitários (Antivírus, Navegadores WEB, Reprodutores de áudio e vídeo, editores de imagens, etc.). Na camada mais externa encontramos os programas aplicativos que podem ser desenvolvidos utilizando-se os recursos da camada anterior e nas linguagens de programação, utilizando as IDEs, os interpretadores ou os compiladores para poderem ser desenvolvidos executados.

Para o desenvolvimento destes programas é que se faz uso das técnicas de construção de algoritmos, de forma a se garantir que os programas serão gerados com um mínimo de erro e poderão ser mantidos, sem dificuldade, por um programador, que não o tenha desenvolvido.

Ao contrário do que muitos pensam, um computador não faz nada sozinho. Ele é uma máquina rápida, que resolve problemas bem definidos e repetitivos, mesmo complexos, mas somente se for bem programado. Ou seja: se temos LIXO na entrada (quer na forma de maus programas ou dados ruins), teremos LIXO na saída (nossos resultados).

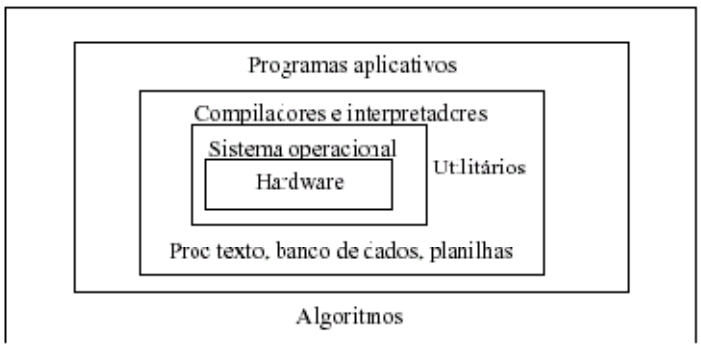


Figura 1.1 – As Camadas de software em um computador.

3. O DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO E O PROGRAMADOR

Um sistema de informação pode ser definido como um sistema baseado em computador que auxilia, automatiza e aperfeiçoa o funcionamento de qualquer atividade através da(o):

- Redução da participação do homem em atividades rotineiras e repetitivas; Aumento da facilidade na coleta e armazenamento de dados e na rapidez de recuperação e manuseio;
- Redução do número de erros produzidos em operações de coleta, arquivamento e recuperação de dados e de informações;
- Aumento de facilidade e flexibilidade na geração de relatórios, entre outros.

Qual o papel do programador e do analista de sistemas no desenvolvimento de sistemas de informação?

Vejamos o Ciclo de Vida de um sistema. Para desenvolvimento de qualquer sistema informatizado de boa qualidade há que se passar pelas seguintes fases:

***1ª Fase*:** Estudo de Viabilidade (Estudos Iniciais das necessidades de um sistema)
***2ª Fase*:** Análise detalhada do sistema (Planejamento com o cliente)
***3ª Fase*:** Projeto preliminar do sistema (Planejamento com os analistas de sistemas)
***4ª Fase*:** Projeto detalhado do sistema (Algoritmos)
***5ª Fase*:** Codificação ou implementação (na linguagem escolhida)
***6ª Fase*:** Testes
***7ª Fase*:** Implementação, operação e manutenção

As três primeiras fases normalmente são de responsabilidade dos Analistas de Sistemas. Nas outras podem trabalhar indistintamente, dependendo da complexidade e da situação, programadores, analistas ou programadores e analistas. A construção dos algoritmos aparece então na fase do projeto detalhado do sistema. Após definidos e criados todos os algoritmos temos de codificá-los na linguagem escolhida. Para essa tarefa o programador deverá conhecer a máquina e o compilador a serem utilizados, e esse assunto será coberto nos próximos módulos.

No desenvolvimento de um sistema, quanto mais tarde um erro é detectado, mais dinheiro e tempo se gasta para repará-lo. Assim, a responsabilidade do programador é maior na criação dos algoritmos do que na sua implementação, pois, quando bem projetados, não se perde muito tempo tendo que refazê-los reimplantá-los e retestá-los, assegurando assim um final feliz e no prazo previsto para o projeto. Entretanto, num projeto, o trabalho de desenvolvimento de algoritmos e programas poderá ter de ser todo refeito se houver problemas nas três primeiras fases, mostrando assim a importância do trabalho do Analista de Sistemas.

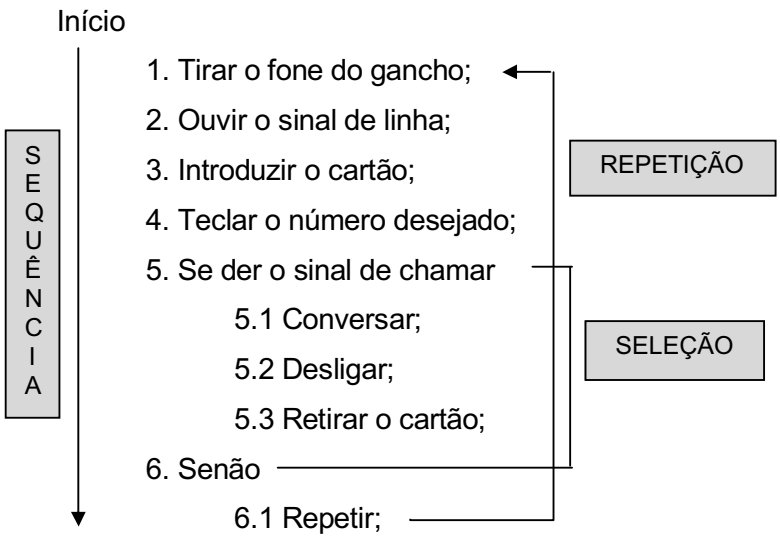
Pode-se encontrar na literatura em informática várias formas de representação das etapas que compõem o ciclo de vida de um sistema. Essas formas de representação podem variar tanto na quantidade de etapas quanto nas atividades a serem realizadas em cada fase.

Como pode-se observar, nesse exemplo de ciclo de vida de um sistema (com sete fases) apresentado acima, os algoritmos fazem parte da quarta etapa do desenvolvimento de um programa. Na verdade, os algoritmos estão presentes no nosso dia-a-dia sem que saibamos, pois uma receita culinária, as instruções de uso de um equipamento ou as indicações de um instrutor sobre como estacionar um carro, por exemplo, nada mais são do que algoritmos.

ALGORITMO

Um Algoritmo é uma sequência de instruções ordenadas de forma lógica para a resolução de uma determinada tarefa ou problema.

Algoritmo não computacional cujo objetivo é a utilização de um telefone público.



Podemos definir como formas básicas para definir uma solução de qualquer problema as etapas de:

SEQUÊNCIA

SELEÇÃO

REPETIÇÃO

Claro que para criar um algoritmo computacional não será tão simples como esse apresentado.

Na informática, o algoritmo é o "projeto do programa", ou seja, antes de se fazer um programa (software) na Linguagem de Programação desejada (Pascal, C, Delphi, Java, PHP, etc.) deve-se fazer o algoritmo do programa. Já um programa, é um algoritmo escrito numa forma compreensível pelo computador (através de uma Linguagem de Programação), onde todas as ações a serem executadas devem ser especificadas nos mínimos detalhes e de acordo com as regras de sintaxe1 da linguagem escolhida.

Um algoritmo não é a solução de um problema, pois, se assim fosse, cada problema teria um único algoritmo. Um algoritmo é um 'caminho' para a solução de um problema e, em geral, existem muitos caminhos que levam a uma solução satisfatória, ou seja, para resolver o mesmo problema pode-se obter vários algoritmos diferentes.

Assim podemos definir que um algoritmo precisa:

1. Ter início e fim;
2. Ser descrito em termos de ações não ambíguas e bem definidas;
3. Que as ações sigam uma sequência ordenada.

1. PSEUDO-CÓDIGO

Os algoritmos são descritos em uma linguagem chamada pseudo-código. Este nome é uma alusão à posterior implementação em uma linguagem de programação, ou seja, quando formos programar em uma linguagem, por exemplo, 'C', estaremos gerando código em 'C'. Por isso os algoritmos são independentes das linguagens de programação. Ao contrário de uma linguagem de programação não existe um formalismo rígido de como deve ser escrito o algoritmo.

O algoritmo deve ser fácil de interpretar e fácil de codificar. Ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação. Utilizaremos em nosso curso o **PORTUGOL** para o estudo dos algoritmos e a Linguagem de Programação 'C' para a criação dos programas.

2. Português Estruturado

O Português Estruturado é uma forma especial de linguagem bem mais restrita que a Língua Portuguesa e com significados bem definidos para todos os termos utilizados nas instruções (comandos).

Essa linguagem também é conhecida como Portugol (junção de Português com Algol, Pseudocódigo ou Pseudolinguagem. O Português Estruturado na verdade é uma simplificação extrema da língua portuguesa, limitada a pouquíssimas palavras e estruturas que têm significado pré-definido, pois deve-se seguir um padrão. Emprega uma linguagem intermediária entre a linguagem natural e uma linguagem de programação, para descrever os algoritmos.

A sintaxe do Português Estruturado não precisa ser seguida tão rigorosamente quanto a sintaxe de uma linguagem de programação, já que o algoritmo não será executado como um programa.

Embora o Português Estruturado seja uma linguagem bastante simplificada, ela possui todos os elementos básicos e uma estrutura semelhante à de uma linguagem de programação de computadores. Portanto, resolver problemas com português estruturado pode ser uma tarefa tão complexa quanto a de escrever um programa em uma linguagem de programação qualquer, só não tão rígida quanto a sua sintaxe, ou seja, o algoritmo não deixa de funcionar porque esquecemos de colocar um ';' (ponto evírgula)por exemplo, já um programa não funcionaria. A Figura a seguir apresenta um exemplo de algoritmona forma de representação de português estruturado.

```
início
    <instruções>
    se <teste> então
        <instruções>
    senão
        <instruções>
    fim_se
fim
```

Figura 2.1: Exemplo de Português Estruturado

3. LINGUAGEM DE PROGRAMAÇÃO

Uma linguagem de programação é uma notação formal para descrição de algoritmos que serão executados por um computador. Como todas as notações formais, uma linguagem de programação tem dois componentes: Sintaxe e Semântica. A sintaxe consiste em um conjunto de regras formais, que especificam a composição de programas a partir de letras, dígitos, e outros símbolos. Por exemplo, regras de sintaxe podem especificar que cada parênteses aberto em uma expressão aritmética deve corresponder a um parênteses fechado, e que dois comandos quaisquer devem ser separados por um ponto e vírgula. As regras de semântica especificam o “significado” de qualquer programa, sintaticamente válido, escrito na linguagem.

4. TIPOS DE LINGUAGENS DE PROGRAMAÇÃO

Existem diversas linguagens de programação, cada uma com suas características específicas e com níveis de complexidade e objetivos diferentes como já vimos anteriormente:

- Linguagem de Máquina — Única compreendida pelo computador. Específica para cada tipo de computador (processador).
- Linguagens de Baixo Nível — Ex.: Assembly
- Linguagens de Alto Nível — Utiliza instruções próximas da linguagem humana de forma a facilitar o raciocínio.
  Ex.: Uso Científico: Fortran
  Propósito Geral: Pascal, C, Basic, Delphi, Java
  Uso Comercial: Cobol, Clipper

5. PROCESSO DE CRIAÇÃO E EXECUÇÃO

Embora seja **teoricamente** possível a construção de computadores especiais, capazes de executar programas escritos em uma linguagem de programação qualquer, os computadores, existentes hoje em dia são capazes de executar somente programas em linguagem de baixo nível, a Linguagem deMáquina.

As Linguagens de Máquina são projetadas levando-se em conta os seguintes aspectos:

· Rapidez de execução de programas;

· Custo de sua implementação;

· Flexibilidade com que permite a construção de programas de nível mais alto.

Por outro lado, linguagens de programação de alto nível são projetadas em função de:

· Facilidade de construção de programas

· Confiabilidade dos programas

O PROBLEMA É: Como a linguagem de nível mais alto pode ser implementada em um computador, cuja linguagem é bastante diferente e de nível mais baixo?

SOLUÇÂO: Através da tradução de programas escritos em linguagens de alto nível para alinguagem de baixo nível do computador.

Para isso existem três tipos de programas tradutores: Montadores, Interpretadores e Compiladores ou simplesmente uma IDE.

MONTADOR Efetua a tradução de linguagem de montagem (Assembly) para a linguagem demáquina.

1. Obtém próxima instrução do Assembly
2. Traduz para as instruções correspondentes em linguagem de máquina
3. Executa as instruções em linguagem de máquina
4. Repete o passo 1 até o fim do programa

INTERPRETADOR Efetua a tradução de uma linguagem de alto nível para linguagem demáquina da seguinte forma:

1. Obtém próxima instrução do código-fonte em linguagem de alto nível
2. Traduz para as instruções correspondentes em linguagem de máquina
3. Executa as instruções em linguagem de máquina
4. Repete o passo 1 até o fim do programa

COMPILADOR Efetua a tradução de todo o código-fonte em linguagem de alto nível para as instruções correspondentes em linguagem de máquina, gerando o código-objeto do programa. Em seguida é necessário o uso de outro programa (Link-Editor) que é responsável pela junção de diversos códigos-objeto em um único programa executável.

IDEs Facilitam a técnica de RAD (de Rapid Application Development, ou "Desenvolvimento Rápido de Aplicativos"), que visa a maior produtividade dos desenvolvedores.

As características e ferramentas mais comuns encontradas nos IDEs são:

- Editor - edita o código-fonte do programa escrito na(s) linguagem(ns) suportada(s) pela IDE;
- Compilador (*compiler*) - compila o código-fonte do programa, editado em uma linguagem específica e a transforma em linguagem de máquina;
- *Linker* - liga (*linka*) os vários "pedaços" de código-fonte, compilados em linguagem de máquina, em um programa executável que pode ser executado em um computador ou outro dispositivo computacional.
- Depurador (*debugger*) - auxilia no processo de encontrar e corrigir defeitos no código-fonte do programa, na tentativa de aprimorar a qualidade de software;
- Modelagem (*modelling*) - criação do modelo de classes, objetos, interfaces, associações e interações dos artefatos envolvidos no software com o objetivo de solucionar as necessidadesalvo do software final.
- Geração de código - característica mais explorada em Ferramentas CASE, a geração de código também é encontrada em IDEs, contudo com um escopo mais direcionado a templates de código comumente

utilizados para solucionar problemas rotineiros. Todavia, em conjunto com ferramentas de modelagem, a geração pode gerar todo ou praticamente todo o código-fonte do programa com base no modelo proposto, tornando muito mais rápido o processo de desenvolvimento e distribuição do software;

- Distribuição (*deploy*) - auxilia no processo de criação do instalador do software, ou outra forma de distribuição, seja discos ou via internet.
- Testes Automatizados (*automated tests*) - realiza testes no software de forma automatizada, com base em scripts ou programas de testes previamente especificados, gerando um relatório, assim auxiliando na análise do impacto das alterações no código-fonte. Ferramentas deste tipo mais comuns no mercado são chamadas robôs de testes.
- Refatoração (*refactoring*) - consiste na melhoria constante do código-fonte do software, seja na construção de código mais otimizado, mais limpo e/ou com melhor entendimento pelos envolvidos no desenvolvimento do software. A refatoração, em conjunto com os testes automatizados, é uma poderosa ferramenta no processo de erradicação de "bugs", tendo em vista que os testes "garantem" o mesmo comportamento externo do software ou da característica sendo reconstruída.

6. ERROS NUM PROGRAMA

Erros de Compilação: Erros de digitação e de uso da sintaxe da linguagem

Erros de Link-Edição: Erro no uso de bibliotecas de subprogramas necessárias ao programa principal.

Erros de Execução: Erro na lógica do programa (algoritmo).

7. CRITÉRIOS DE QUALIDADE

Refere-se à precisão das informações manipuladas pelo programa, ou seja, os resultados gerados pelo processamento do programa devem estar corretos, caso contrário o programa simplesmente não tem sentido.

Clareza: Refere-se à facilidade de leitura do programa. Se um programa for escrito com clareza, deverá ser possível a outro programador seguir a lógica do programa sem muito esforço, assim como o próprio autor do programa entendê-lo após ter estado um longo período afastado dele. (Comentários no código)

Simplicidade: A clareza e precisão de um programa são normalmente melhoradas tornando as coisas o mais simples possível, consistentes com os objetivos do programa. Muitas vezes torna-se necessário sacrificar alguma eficiência de processamento, de forma a manter a estrutura do programa mais simples.

Eficiência: Refere-se à velocidade de processamento e a correta utilização da memória. Um programa deve ter desempenho SUFICIENTE para atender as necessidades do problema e do usuário, bem como deve utilizar os recursos de memória de forma moderada, dentro das limitações do problema.

Modularização: Durante a fase de projeto, a solução do problema total vai sendo fatorada em soluções de subproblemas, o que permite, geralmente, dividir o problema de forma natural em módulos com subfunções claramente delimitadas, que podem ser implementados separadamente por diversos programadores de uma equipe, ou seja, a **modularização** consiste no particionamento do programa em módulos menores bem identificáveis e com funções específicas, de forma que o conjunto desses módulos e a interação entre eles permite a resolução do problema de forma mais simples e clara.

Generalidade: É interessante que um programa seja tão genérico quanto possível de forma a permitir a reutilização de seus componentes em outros projetos.

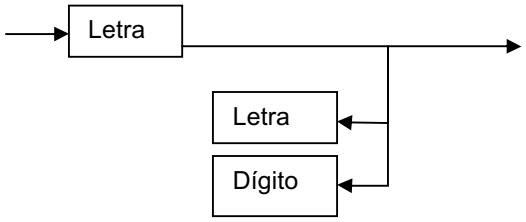
METODOLOGIA DE SOLUÇÃO

1. Entender o problema;
2. Formular um esboço da solução;
3. Fazer uma primeira aproximação das variáveis necessárias;
4. Rever os passos originais, detalhando;
5. Se o algoritmo estiver suficientemente detalhado, testar com um conjunto de dados significativos;
6. Implementar numa linguagem de programação.

COMANDOS BÁSICOS

1. IDENTIFICADORES, CONSTANTES, VARIÁVEIS E TIPOS BÁSICOS

• **Identificadores:** Representam os nomes escolhidos para rotular as variáveis, procedimentos e funções, normalmente, obedecem as seguintes regras:



1. O primeiro caractere deve ser uma letra
2. Os nomes devem ser formados por caracteres pertencentes ao seguinte conjunto: {a,b,c,..., z,A,B,C,...Z,0,1,2,...,9,_}
3. Os nomes escolhidos devem explicitar seu conteúdo.

EX: A, B1, BC3D,SOMA, CONTADOR;

Obs.: Um exemplo de identificador inválido seria 2AB ou qualquer outro iniciado por um dígito

• **Constante ("constant")** – Uma constante (Const), como sugere o nome, é umidentificador que armazena um valor fixo e imutável, durante a execução de um algoritmoou programa. Podemos associá-lo a uma posição de memória (endereço) que tem umconteúdo fixo. Este conteúdo poderá ser um número (real ou inteiro), uma cadeia decaracteres (texto) ou um valor lógico (será definido abaixo).

• **Variável ("variable")** – Uma variável (Var) é um identificador que, como sugere onome, possui o conteúdo variável durante a execução de um algoritmo ou programa.Podemos associar uma variável a uma posição da memória (endereço) e poderemosarmazenar (guardar) neste endereço qualquer valor do conjunto de valores de um tipobásico associado a ela. Uma variável pode assumir vários valores diferentes ao longo daexecução do programa, mas, em um determinado momento, possui apenas um valor.Unidades básicas de armazenamento das informações a nível de linguagens deprogramação. Os tipos de dados e variáveis utilizados dependem da finalidade dosalgoritmos, mas, podemos definir alguns, pelo fato de serem largamente utilizados eimplementados na maioria das linguagens:

Esse tipo básico poderá ser:

***INTEIRO* ("int, short int ou long int"):** qualquer número inteiro, negativo, nulo ou positivo.
Ex: -2, -1, 0...
Operações: soma(+), subtração(-), multiplicação(*), divisão inteira(/), resto(%) e comparações.

***REAL* ("float ou double"):** qualquer número real, negativo, nulo ou positivo.
Ex: 2.5, 3.1
Operações: soma(+), subtração(-), multiplicação(*), divisão exata(/) e comparações.

***CARACTER* ("char"):** qualquer conjunto de caracteres alfanuméricos.
Ex: A, B, "ABACATE"
Operações: comparações

***TEXTO OU CADEIA DE CARACTERES* ("string"):**uma variável deste tipo poderá armazenaruma cadeia de caracteres de qualquer tamanho. Caso seja imprescindível para o entendimentopode-se acrescentar, entre parênteses, a quantidade máxima de caracteres. (Exemplo: texto (10)).

Obs.: Os textos deverão ser representados sempre entre apóstrofes para que não se confundamcom os valores numéricos. Veja que o inteiro 5, é diferente do texto '5'.

***LÓGICO* (“boolean”):** tipo especial de variável que armazena apenas os valores V e F, onde Vrepresenta VERDADEIRO e F FALSO.

Ex: e, ou, não

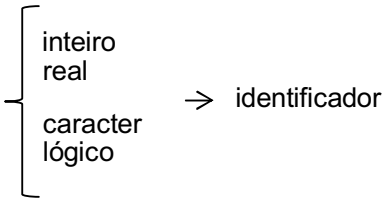
Operações: Verdadeiro ou Falso

2. DECLARAÇÃO DE VARIÁVEIS

Consiste na definição dos nomes e valores das constantes e dos nomes e tipos dasvariáveis que serão utilizadas pelos algoritmos, previamente à sua utilização, incluindocomentário, quando se fizerem necessários.

Na maioria das linguagens de programação, quando o computador está executando umprograma e encontra uma referência a uma variável ou a uma constante qualquer, se esta nãotiver sido previamente definida, ele não saberá o que fazer com ela. Da mesma forma, umprogramador que estiver implementando um algoritmo, em alguma linguagem de programação,terá o seu trabalho simplificado se todas as constantes e variáveis referenciadas no algoritmotiverem sido previamente declaradas. As constantes são declaradas antes das variáveis. Vejamosos formatos da declaração e alguns exemplos.

O significado da declaração de variáveis corresponde à criação de locais na memória rotuladacom o nome da variável (identificador) e marcada com o tipo de valores que ela pode conter. Paraque os programas manipulem valores, estes devem ser armazenados em variáveis e para isso,devemos declará-las de acordo com a sintaxe:



Ex:
Inteiro X1;
obs.: X1 é o nome de um local de memória que só pode conter valores do tipo inteiro
real SOMA, MÉDIA;
caractere frase, nome;
inteiro X1;
real A,B;
lógico TEM;

3. PALAVRAS RESERVADAS

São palavras que terão uso específico no nosso pseudo-código e que não deverão serusadas como identificadores, para não causar confusão na interpretação.

Exemplo: Algoritmo, Programa, Bloco, Procedimento, Inteiro, Real, Texto, Const,Var, Tipo, Início, Imprima, Se, Então, Senão, Enquanto, Repita, Variando,Faça, Caso, Até, Vetor, Matriz, Registro, Fim, Execute, Procedimento, Função, etc....

O significado de cada um desses termos será visto e entendido nos itens e capítulos que seseguem.

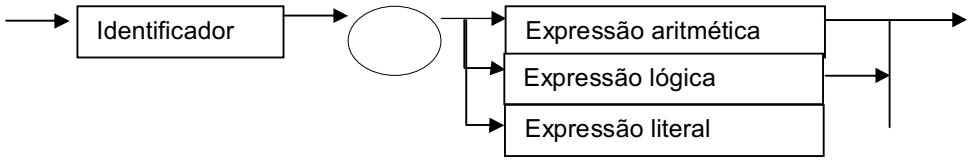
COMANDO SIMPLES
É uma instrução simples.
leia(x);

COMANDO COMPOSTO
Um grupo de comandos simples que executam alguma tarefa.
Início
leia(x);
y = 2*x;
fim.

4. OPERADORES

Na solução da grande maioria dos problemas é necessário que as variáveis tenham seus valores consultados ou alterados, para isto, devemos definir um conjunto de **OPERADORES,** sendo eles:

3.1. OPERADOR DE ATRIBUIÇÃO



Ex: O valor da expressão é atribuído ao identificador (variável).
X = 2; y = 5-x;
Este comando permite que se forneça ou altere o valor de uma determinada variável, onde o tipo desse valor seja compatível ao tipo de variável na qual está sendo armazenado, de acordo com o especificado na declaração.
NUM = 8 {A variável NUM recebe o valor 8}
NOME = "Guilherme" {A variável NOME recebe o valor 'Guilherme'}
CONT = 0
AUXNOME = NOME {A variável AUXNOME recebe o conteúdo da variável NOME}
ACHOU = falso {A variável ACHOU recebe o valor falso}

3.2. OPERADORES ARITMÉTICOS:

+ = Adição	% = Resto da divisão de inteiros 5%2=1
* = Multiplicação	a**b = Exponenciação a^b
- = Subtração ou inverso do sinal.	SINAL(A) - Fornece o valor -1, +1 ou zero conforme o valor de A seja negativo, positivo ou igual a zero.
/ = Divisão	TRUNCA(A) - A parte inteira de um número Fracionário.
/ = Quociente da divisão de inteiros 5/2=2	ARREDONDA(A) - Transforma por arredondamento, um número fracionário em inteiro.

Obs. FUNÇÕES PRIMITIVAS: As funções mais comuns de matemática são também definidas e válidas no PORTUGOL.

Exemplos:
LOG (X), {dá o logaritmo na base 10 de X}
SEN (X), {dá o seno de X}
ABS (X), {dá o valor absoluto de X}
INT (X), {dá a parte inteira de um real X}
ARREDONDA (x), {arredonda um real para inteiro}
RAIZ (X), {dá a raiz quadrada de X} etc.

3.3. OPERADORES RELACIONAIS:

São utilizados para relacionar variáveis ou expressões, resultando num valor lógico (Verdadeiro ou Falso), sendo eles:

== igual	!= diferente de
<menor	>maior
<= menor ou igual	>= maior ou igual

Exemplo:
NUM = 3
NOME = 'DENISE'
NUM > *5* é falso (0) {3 não é maior que 5}
NOME < 'DENIZ' é verdadeiro (1) {DENISE vem antes de DENIZ}

(5 + 3) > = 7 é verdadeiro
NUM != (4 – 1) é falso {lembre-se que NUM “recebeu” 3}

3.4. OPERADORES LÓGICOS:

Os operadores lógicos permitem que mais de uma condição seja testada em uma única expressão, ouseja, pode-se fazer mais de uma comparação (teste) ao mesmo tempo.

Operação	Operador
Negação	não
Conjunção	e
Disjunção (não exclusiva)	ou
Disjunção (exclusiva)	xou (lê-se: “ou exclusivo”)

Note que a tabela acima, apresenta os operadores lógicos já ordenados de acordo com suasprioridades, ou seja, se na mesma expressão tivermos o operador **ou** e o operador **não**, por exemplo,primeiro devemos executar o **não** e depois o **ou**.

De uma forma geral, os resultados possíveis para os operadores lógicos podem ser vistos natabela abaixo, conhecida como **Tabela Verdade**:

A	B	A e B	A ou B	não A	A xou B
F	F	F	F	V	F
F	V	F	V	V	V
V	F	F	V	F	V
V	V	V	V	F	F

Exemplos de testes utilizando operadores lógicos:

Expressão	Quando eu não saio?
Se chover **e** relampejar, eu não saio.	Somente quando chover e relampejar ao mesmo tempo (apenas 1 possibilidade).
Se chover ***ou***relampejar, eu não saio.	Somente quando chover, somente quandorelampejar ou quando chover e relampejar aomesmo tempo (3 possibilidades).
Se chover ***xou***relampejar, eu não saio.	Somente quando chover, ou somente quandorelampejar (2 possibilidades).

1. Se (**salario > 180) e** (**salário < 800)** Então
   Escrever (**'**Salário válido para financiamento**')**
   Senão
   Escrever (**'**Salário fora da faixa permitida para financiamento**')**
   FimSe.

2. Se (**idade < 18) ou (idade > 95)** Então
   Escrever (**'**Você não pode fazer carteira de motorista**')**
   Senão
   Escrever (**'**Você pode possuir carteira de motorista**')**
   FimSe

3. Se (**idade > = 18) e (idade < = 95) e (aprovado_exame = 'sim')** Então
   Escrever (**'**Sua carteira de motorista estará pronta em uma semana**')**
   Senão
   Escrever (**'**Você não possui idade permitida *ou* não passou nos testes**')**
   FimSe.

3.5. PRIORIDADE DE OPERADORES:

Durante a execução de uma expressão que envolve vários operadores, é necessário a existência de prioridades, caso contrário poderemos obter valores que não representam o resultado esperado.

A maioria das linguagens de programação utiliza as seguintes prioridades de operadores:

1º - Efetuar operações embutidas em parênteses "mais internos"
2º - Efetuar Funções
3º - Efetuar multiplicação e/ou divisão
4º - Efetuar adição e/ou subtração
5º - Operadores Relacionais
6º - Operadores Lógicos

Ou seja,
Primeiro: Parênteses e Funções
Segundo: Expressões Aritméticas
1) +, - (Unitários)
2) **
3) *, /
4) +, - (binário)
Terceiro: Comparações
Quarto: Não
Quinto: e
Sexto: ou

OBS: O programador tem plena liberdade para incluir novas variáveis, operadores ou funções para adaptar o algoritmo as suas necessidades, lembrando sempre, de que, estes devem ser compatíveis com a linguagem de programação a ser utilizada.

5. EXERCÍCIOS INTRODUTÓRIOS – I

1. O resultado da expressão 3 + 6 * 13 = X?

2. Sendo A,B,C e D variáveis do tipo inteiro, cujos conteúdos são: A=3, B=4, C=13 e D=4. Quais os valores fornecidos por cada uma das expressões aritméticas abaixo?

a) 100*(B/A)+C
b) A%5 - D/2
c) trunca(0,3+5)*2
d) (C+D)/C**2
e) B%(A+1)
f) sinal(D-C)

3. Qual é a primeira operação executada em cada um dos comandos abaixo?

a) X+Y-Z;
b)A+B/C**2;
c) JOÃO+JOSÉ/JOEL;
d) X+Y+B**2+R*3

4. Avalie a seguinte expressão aritmética: 6+4/2*3. O resultado desta expressão é 15; *12*; 24?

5. Qual o resultado das seguintes expressões

a) 1 + 7 * 2 ** 2 - 1 = X
b) 3 * (1 - 2) + 4 * 2 = X

6. Avalie as expressões abaixo sabendo que X = 2 e Y = 3

a) 3 * y - x ** 2 * (7 - y ** 2)
b) 2 * x * (3 - y) / (x ** 2 - y)

7. Identifique as entradas e saídas das expressões abaixo. Escreva-as em forma computacional

a) r=2y²+3zb)$x = \frac{-b+\sqrt{b^2-4ac}}{2a}$

6. COMANDOS DE ENTRADA E SAÍDA

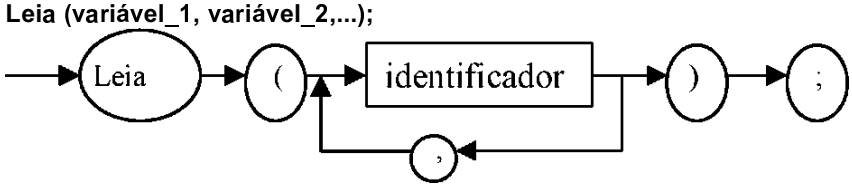
No algoritmo é preciso representar a troca de informações que ocorrerá entre o mundo da máquina e o nosso mundo, para isso, devemos utilizar comandos de entrada e saída, sendo que, no nível de algoritmo esses comandos representam apenas a entrada e a saída da informação, independe do dispositivo utilizado (teclado, discos, impressora, monitor,...), mas, sabemos que nas linguagens de programação essa independência não existe, ou seja, nas linguagens de programação temos comandos específicos para cada tipo de unidade de Entrada/Saída.

6.1. Comando de Entrada de Dados

Para que possamos obter dados do meio exterior para uso do computador (memória principal), estes têm de vir através dos dispositivos de entrada. Da mesma forma, as informações que são produzidas, têm de ser levadas ao meio externo (um arquivo, uma impressora, uma tela etc.) através de um dispositivo de saída. Para isso, utilizamos dois comandos assim definidos:

Comando **Leia** (**scanf** no c): lê, do meio externo, a próxima informação disponível para leitura e armazena na(s) variável(eis) discriminada(s) após o comando, entre parênteses. Mais tarde aprenderemos como especificar a leitura de um dado que está armazenado em um arquivo e de que arquivo está sendo lido o dado.

Leia (variável_1, variável_2,...);

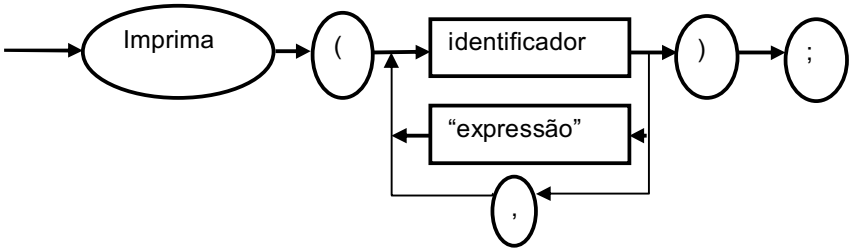


Ex: leia (v); O valor da variável (v) é dado por um dispositivo de entrada (teclado).

3.6. Comando de Saída de Dados

Comando **Imprima** (**printf** no c): Imprime (na tela ou na impressora) o conteúdo da(s) variável(eis) especificada(s) após o comando, entre parênteses. Não será preocupação nossa a formatação de relatórios, mas o comando permite a impressão de texto (entre“), se for necessária para clareza ou especificação do que está sendo impresso.

Imprima (“o valor de x” x);



Ex: imprima (x); O valor atual da variável (x) é informado para um dispositivo de saída(monitor).

7. ESTRUTURA DE UM ALGORITMO EM PSEUDO-CÓDIGO E ‘C’

As estruturas de controle introduzidas nesta unidade facilitarão a expressão de algoritmos e permitirão que exemplos mais significativos sejam apresentados e propostos, visando o aperfeiçoamento do desenvolvimento do processo de aprendizado da construção de algoritmos.

Descreveremos a seguir os comandos básicos de controle do PORTUGOL. Serão introduzidos paralelamente o programa equivalente em linguagem ‘C’ para uma melhor compreensão.

• **PORTUGOL**

Algoritmo
Início
<declarações de variáveis>;

```
<comandos>;
Fim.
```

• **C**

```
#include <stdio.h>
main () /* Um Primeiro Programa */
{
    printf ("Meu primeiro programa\n");
}
```

Comentário de códigos: Quando fazemos um programa, uma boa ideia é usar comentáriosque ajudem a elucidar o funcionamento do mesmo. Os comentários devem vir entre /* */

Ex: **/* Um Primeiro Programa */**
O compilador C desconsidera qualquer coisa que esteja começando com**/*** e terminando */.

DOCUMENTAÇÃO
Alguns princípios básicosdeverão ser usados desde a criação do primeiro algoritmo:

- Ao criar as constantes e variáveis, utilizar nomes significativos e comentar, se necessário for;
- Utilizar identação (três espaços como sugestão), para mostrar a estrutura lógica e sequência de comandos, quando usando o PORTUGOL e 'C'.
- Utilizar parênteses para evitar ambiguidade nas expressões;

ESTRUTURAS BÁSICAS DE CONTROLE

No capítulo anterior, foram apresentados exemplos das três estruturas necessárias para representação da solução de qualquer algoritmo: a sequência, a seleção e a repetição. Em se tratando de algoritmo para computadores, as três também formam a base das estruturas de controle de execução. Vejamos como formalizaremos a representação e o uso dessas estruturas no PORTUGOL e na Linguagem 'C'.

1. SEQUÊNCIA

Grupo de comandos que são executados um após o outro.

• **PORTUGOL**

```
Início
  comando 1;
  comando 2;
  .
  .
  comando n;
Fim.
```

• **C**

```
#include <stdio.h>
main () /* Um Primeiro Programa */
{
    printf ("Meu primeiro programa\n");
}
```

2. SELEÇÃO

Também chamada de estrutura de decisão ou de processamento condicional, a estrutura de seleção é utilizada quando a execução de um comando (ou uma sequência de comandos) depende de um teste anterior (uma ou mais comparações). A seleção pode ser simples ou composta.

2.1. SELEÇÃO SIMPLES

Quando a execução de um comando (ou de uma sequência de comandos) depender de uma condição verdadeira, e não há comandos a executar se a condição for falsa.

• **PORTUGOL**

```
Se <condição> Então
  (comandos);
Fim Se;
```

• **C**

```
if (num==10) //Condição
  {
printf ("O numero e igual a 10."); //Comando
}
```

Onde a CONDIÇÃO poderá ser também uma expressão lógica.
Exemplo1: leia um número inteiro e o imprima se ele for diferente de 0 (ZERO)
Exemplo2:

```
Se (a<5) Então
Imprima (“o valor de a é ” a);
Fim Se;
```

2.2. SELEÇÃO COMPOSTA

Quando se executa um comando (ou sequência de comandos) se uma condição é verdadeira, e se executa um outro comando (ou sequência de comandos) se a condição é falsa.

• **PORTUGOL**

```
Se <condição> Então
  (comandos);
Senão
  (comandos);
Fim Se;
```

• **C**

```
if (num==10) // Condição
  {
printf ("O numero e igual a 10."); // Comando
}
else
  {
printf ("O numero e diferente de 10."); // Comando
  }
```

2.3. ANINHAMENTO DE SELEÇÕES

A estrutura de Seleção permite o aninhamento, ou seja, o comando a ser executado dentro de uma seleção (por exemplo, no "Senão") pode ser outra seleção. Outro aninhamento poderá ocorrer também com esta última seleção e assim por diante. Nos casos de vários aninhamentos subsequentes, uma boa identação será fundamental para o entendimento do algoritmo quando utilizando pseudo-código.

• **PORTUGOL**

```
Início
  -
  Se CONDIÇÃO_A Então {V}
    comando1;
  Senão {F}
    Se CONDIÇÃO_B Então
    comando2;
    Senão
    comando3;
    Fim Se;
  Fim se;
  -
Fim.
```

• **C**

```
if (num <= 10)
{
 if (num == 10)
printf ("O numero e igual a 10.");
     else
       printf (“O numero e menor que 10”);
}
else if (num <= 20)
  {
printf ("O numero e maior que 10 e menor que 20.");
  }
```

3. ESTRUTURAS DE REPETIÇÃO - LAÇOS(LOOPS)

3.1. LAÇO ENQUANTO(WHILE)

A estrutura de repetição (enquanto) é utilizada quando um conjunto de comandos deve ser executado repetidamente, enquanto uma determinada condição (expressão lógica) permanecer verdadeira. Dependendo do resultado do teste da condição, o conjunto de comandos poderá não ser executado nem uma vez (se for falsa no primeiro teste), ou será executado várias vezes (enquanto for verdadeira). Chama-se a isso um laço ("loop").

Da mesma forma que a estrutura de seleção, ela permite o aninhamento de repetições, ou seja, a existência de uma estrutura de repetição dentro de outra. Poderão haver também aninhamentos de seleções dentro de estruturas repetitivas e vice-versa. Dois cuidados ao criar estruturas de repetição (enquanto):

1. Inicializar a(s) variável(eis) que controla(m) o laço antes do início do laço;
2. Inicializar a(s) variável(eis) que controla(m) o laço dentro do laço (seja por leitura ou por atribuição), pois se isto não for feito cairemos no que chamamos um laço infinito e de lá o nosso programa não sairá.

• **PORTUGOL**

```
Início
Inteiro x;
x = 2;
Enquanto (x<10) Faça
Imprima (x);
x=x+1;
Fim Enquanto;
Imprima(x);
Fim.
```

• **C**

```
#include <stdio.h>
main ()
{
     int X;
     X = 2;
while (X < 10)
{
          printf (“%d”, &X);
          X = X + 1;
}
     printf (“%d”, &X);
}
```

Exercício: faça um trecho de algoritmo para ler e imprimir o nome e a idade de várias pessoas, até encontrar alguém com 65 anos, quando deverá ser impresso, além do nome, uma mensagem informando o fato.

3.2. CONTROLADOS POR CONTADOR

Uma variável é fornecida com o n.º de vezes que será repetido o laço.

```
Algoritmo Abono_por_Contador
  Início
    Leia (Numero_de_Funcionários);
    Contador = 0;
```

```
    Enquanto (Contador < Número_de_Funcionários) Faça
      Leia (Nome);
      Contador = Contador+1;
    Fim Enquanto;
Fim.
```

3.7. REPETIÇÃO COM TESTE NO FINAL

Quando se deseja executar a série de comandos uma vez pelo menos, pode se fazer o teste no final. Essa variação tem um uso bastante efetivo na validação de dados de entrada, pelo teclado, mas pode ser sempre substituída por um enquanto. Uma vantagem do repita é que não é preciso inicializar a(s) variável(eis) de controle do laço antes de entrar no mesmo. Deve-se, contudo, ter o cuidado de modificá-la(s) dentro do laço para que não caiamos em um laço infinito.

Executa uma instrução e faz um teste lógico. Dependendo da resposta, fica repetindo o processoenquanto o teste lógico for Verdadeiro.

• **PORTUGOL**

```
repita
  c1;
  c2;
  .
  .
  cn;
enquanto<condição>;
```

•**C**

```
do
  {
c1;
c2;
.
  .
cn;
  } while (condição);
```

Os comandos c1, c2,....,cn são executados pelo menos uma vez. Quando a condição é encontrada, ela é testada, se for falsa o comando seguinte será executado, se for verdadeira, os comandos c1, c2,...,cn são reexecutados enquanto a condição for verdadeira.

O comando repita-enquanto é equivalente ao comando enquanto, conforme será mostrado no exemplo abaixo.

- **Repita – enquanto (do – while)**

```
Início
  Inteiro x;
x = 2;
  Repita
    Imprima(x);
x = x+1;
Enquanto (x<=10);
  Imprima(x);
Fim.
```

- **Enquanto (while)**

```
Início
  Inteiro: x;
  x←2;
  Enquanto x<10 faça
    Imprima (x);
    x←x+1;
  Fim Enquanto;
  Imprima (x);
Fim.
```

Numa estrutura **Enquanto**, os comandos são executados 0 ou mais vezes. Numa estrutura **Repita**, os comandos são executados pelo menos uma vez.

3.8. ESCAPE DO LAÇO – Abandone (break)

Comando Abandone: Força uma saída de um laço de repetição.

• **PORTUGOL**
abandone

• **C**
break

Exemplo:

```
Enquanto (a>b) faça
  Leia(a,b);
  Imprima(a,b);
  Se b==0 Então
abandone;
Senão a=a/b;
Fim Enquanto;
```

O comando abandone só tem sentido dentro de um comando de repetição (enquanto, repita, para). Além disso, estará sempre associado ao teste de uma condição com comando se.

• **Significado do comando abandone**: quando o abandone é encontrado, o próximo comando a ser executado é o primeiro comando logo após o fim do comando de repetição mais interno onde aparece o abandone.

3.9. Sinalizador ('Flags')

O sinalizador ou 'flag' é um valor que, quando lido, sinaliza a saída de um laço, o fim de um programa, o fim de uma lista de dados, ou o fim de um arquivo. Para identificá-lo iremos defini-lo como uma constante nos nossos algoritmos.

Exemplo: faça um trecho de algoritmo utilizando o PORTUGOL que leia e imprima uma lista de números inteiros terminada pelo número 999.

```
Inicio
  Constante FLAG = 999;
  Inteiro NUM;
  Leia (NUM)
  Enquanto (NUM <> FLAG) faça
    Imprima (NUM);
    Leia (NUM);
  Fim Enquanto;
Fim.
```

4. EXERCÍCIOS INTRODUTÓRIOS – II

1) Quais valores serão impressos após a execução do seguinte trecho de algoritmo

```
I=0;
Enquanto I<20 Faça
    Imprima (I,I**2);
    I=I+1;
Fim Enquanto;
Imprima (“I=”,I);
Imprima (“UFA! CHEGUEI AQUI”); /*CUIDADO*/
```

2) Dado o algoritmo abaixo, que calcula o volume de uma esfera de raio R:

```
Início
  Real VOLUME, PI, R;
```

```
Constante PI = 3,1416;
R=0;
Enquanto R<= 6 Faça
    VOLUME=4/3*PI*(R**3);
    Imprima (R,VOLUME);
    R=R+2;
Fim Enquanto;
Fim.
```

Completar os demais valores que serão impressos:

R	VOLUME DA ESFERA

10) Identifique o tipo de cada uma das constantes abaixo:

a) 21 b) "BOLA" c) "VERDADEIRO"

d) 0,21*10² e) falso

5. REGRAS PRÁTICAS PARA A CONSTRUÇÃO DE ALGORITMOS LEGÍVEIS

1- Procure incorporar comentários no algoritmo, pelo menos para descrever o significado das variáveis utilizadas.

Exemplo: /* Cálculo da média */

2- Escolha nomes de variáveis que sejam significativos, isto é, que traduzam o tipo de informação a ser armazenada na variável.

Exemplo: NOTA
MATRÍCULA

3- Procure alinhar os comandos de acordo com o nível a que pertençam, isto é, destaque a estrutura na qual estão contidos.

```
Exemplo:
B=A*C;
Enquanto P==1 Faça
P=P+1;
F=X+NOTA/2;
Fim Enquanto;
Imprima (F,NOTA);
```

4- Tenha em mente que seus algoritmos deverão ser lidos e entendidos por outras pessoas (e por você mesmo) de tal forma que possam ser corrigidos, receber manutenção e serem modificados.

5- Escreva comentários no momento em que estiver escrevendo o algoritmo. Um algoritmo não documentado é um dos piores erros que um programador pode cometer.

6- Use comentários no início para explicar o que o algoritmo faz. Alguns comentários seriam;

- Uma descrição do que faz o algoritmo
- Como utilizá-lo
- Explicação do significado das variáveis mais importantes
- Estrutura de dados utilizada
- Os nomes de quaisquer métodos especiais utilizados
- Autor
- Data da escrita

7- Utilize espaços em branco para melhorar a legibilidade. Espaços em branco, inclusive linhas em branco são valiosíssimos para melhorar a aparência de um algoritmo.

8- Escolha nomes representativos para suas variáveis.

9- Um comando por linha é suficiente

- O algoritmo fica mais legível
- O algoritmo fica mais fácil de ser depurado.

Exemplo:
A=14,2;I=1;enquanto I<10 faça X=X+1;K=I*K;I=I+1; fim enquanto
O mesmo exemplo com cada comando em uma linha
A=14,2;
I=1;
Enquanto I<10 Faça
X=X+1;
K=I*K;
I=I+1;
Fim Enquanto;

10- Utilize parênteses para aumentar a legibilidade e prevenir-se contra erros.

Com poucos parênteses	Com parênteses extras
A*B*C/(C*D*E)	(A*B*C)/(C*D*E)
A**B**C	(A**B)**C
A+B<C	(A+B)<C

11- Toda vez que uma modificação for feita no algoritmo, os comentários devem ser alterados.

6. UMA REFLEXÃO ANTES DE RESOLVER OS PROBLEMAS

Passo1 - Leia cuidadosamente a especificação do problema até o final.
Passo 2 - Caso não tenha entendido o problema, pergunte.
Passo 3- Levantar e analisar todas as saídas exigidas na especificação do problema (impressões).
Passo 4- Levantar e analisar todas as entradas citadas na especificação do problema (leituras).
Passo 5- Verificar se é necessário gerar valores internamente ao algoritmo e levantar as variáveis necessárias e os valores iniciais de cada uma.
Passo 6- Levantar e analisar todas as transformações necessárias, dadas as entradas e valores gerados internamente, para produzir as saídas especificadas.
Passo 7- Testar cada passo do algoritmo.
Passo 8 - Avaliação geral, elaborando o algoritmo e revendo comentários.

7. EXERCÍCIOS COMPLEMENTARES - I

1. Faça um algoritmo que leia quatro idades e calcule a média das mesmas.

2. Altere o algoritmo de media de idade para ler também os nomes e ao final mostrar a mensagem com os três nomes mais a média de suas idades.

3. Faça um algoritmo que faça a soma dos números inteiros de 1 a 18.

4. Faça um programa que leia 4 números inteiros e apresente:

• Média dos ímpares

• Maior número par

• Diferença do maior menos o menor número

5. Faça um programa que leia o nome e a idade de 3 pessoas e apresente:

• Maior idade

• Nome da pessoa mais nova

• Média das idades

7. Faça um programa que leia a medida do lado de um quadrado e calcule e apresente a área e o perímetro desta figura. Obs: Perímetro é a soma de todos os lados de uma figura geométrica.

8. Faça um programa que leia o raio de uma circunferência e calcule e apresente sua a área e perímetro.

9. Faça um programa que leia o valor dos lados de um triângulo retângulo e calcule e apresente a sua área.

Área = (base*altura)/2

ALGORÍTMOS BASEADOS EM ESTRUTURAS DE DADOS HOMOGÊNEAS:

1. VETORES E MATRIZES

Antes de tratarmos das estruturas de dados homogêneas, vejamos algumas novas estruturas de controle, que faremos uso quando trabalharmos com vetores e matrizes.

2. REPETIÇÃO COM VARIÁVEL DE CONTROLE – PARA (for)

Repete uma instrução um pré-determinado número de vezes.
para **v** de **i** até **f**passo **p** faça
onde:
v: variável de controle.
i: valor inicial de v.
f: valor final de v.
p: valor do incremento de v.
Sintaxe do comando:

• **PORTUGOL**

```
Para v de i até f passo p Faça
    C1;
    C2;
    .
    .
    Cn;
Fim Para;
```

• **C**

```
for (v = 0; v < 100;v++) /* v = 0; enquanto v for menor que 100 faça; v incremento + 1*/
  {
    printf ("%d", &v)
  }
```

Significado do comando: v, i, f, p são variáveis quaisquer e que, de acordo com as regras da programação estruturada, não devem ser modificadas nos comandos C1, C2, . . ., Cn. O comando **para** é, na verdade, o comando **enquanto** utilizando-se uma variável de controle, escrito numa notação compactada. Neste caso existirá sempre uma inicialização da variável de controle, um teste para verificar se a variável atingiu o limite e um acréscimo na variável.

Exemplo para comparação entre **Enquanto** e **Para**

```
Início
    Inteiro X;
    X = 1 /*inicialização*/
    Enquanto (X <=10) Faça /*teste*/
        Leia(v[X]);
X = X +1; /*acréscimo*/
    Fim Enquanto;
Fim.
```

Equivale a:

```
Início
    Inteiro X;
    Para X de 1 até 10 Passo 1 Faça
        Leia(v[X]);
    Fim Para
Fim.
```

Nota:

1) Assim como no comando enquanto, se o valor inicial (i) já for superior ao limite (l), os comandos C1, C2,...,Cn não serão executados.

2) Toda vez que o ***fim para*** é encontrado, a variável de controle (i) é incrementada pelo passo p e o teste (v<=l) é feito.

3) O valor da variável de controle torna-se indefinido assim que é executado o comando para.

Neste aspecto o comando **para** é diferente do comando **enquanto**. Por exemplo, com o trecho de algoritmo seguinte:

```
Para X de 1 até 10 Passo 1 Faça
Imprima (X);
Fim Para;
Imprima (X);
```

serão impressos dentro do comando **para**: 1,2,3,4,5,6,7,8,9,10 e fora do comando **para** será impresso o valor 11.

Quando o passo (p) for igual a 1, não será necessário escrever esta especificação no comando.
No exemplo acima teríamos:

```
Para X de 1 até 10 Faça
Imprima (X);
Fim Para;
```

3. SELEÇÃO DENTRE AS MÚLTIPLAS ALTERNATIVAS - CASO (CASE)

Em nossos algoritmos, quando uma variável ou expressão aritmética puder assumir váriosvalores diferentes, e a execução do programa for diferente dependendo do valor obtido ouassumido por esta variável ou expressão, poderemos utilizar a estrutura que se segue, ao invésde vários "Se" aninhados.

• **PORTUGOL**

```
Início
Conforme
Caso IDENTIFICADOR = Valor_1
comando1;
Caso IDENTIFICADOR = valor_2
comando2;
Caso IDENTIFICADOR = valor_3
comando3;
-
Senão
comandoN;
Fim Conforme;
-
-
Fim.
```

• **'C'**

```
#include <stdio.h>
main ()
    {
char Ch;
do
{
printf ("\n\nEscolha um:\n\n"); // \n Nova linha
printf ("\t(1)...Mamao\n"); // \t Tabulação horizontal ("tab")
printf ("\t(2)...Abacaxi\n");
printf ("\t(3)...Laranja\n\n");
fflush(NULL);
                scanf("%c",&Ch);
            } while ((Ch!='1')&&|(Ch!='2')&&(Ch!='3'));
        switch (Ch) // Conforme
            {
case '1': // Caso
```

```
printf ("\t\tVoce escolheu Mamao.\n");
break;
case '2':
printf ("\t\tVoce escolheu Abacaxi.\n");
break;
case '3':
printf ("\t\tVoce escolheu Laranja.\n");
break;
}
}
```

1. Após obter-se um valor verdadeiro para um dos casos, cessam os testes dos casos e o próximo comando a ser executado é o que vem a seguir do Fim Conforme.

Esta estrutura também pode ser utilizada testando-se as condições ao invés de valores, paradeterminar qual o comando a ser executado após o teste de cada condição (Ver exemplo b). Emprogramação, uma aplicação sempre interessante desta estrutura é no tratamento de Menus deOpções, quando para cada seleção feita pelo usuário do programa, haverá uma sequência deações diferentes a ser executada. Outra aplicação éna impressão de diferentes mensagens deerro.

***Exemplos*:**

a) Início /*trecho de algoritmo para tratamento de três possíveis erros conhecidos na execução deum programa*/

```
Conforme
Caso ERRO = 1
Imprima (MSG1);
Caso ERRO = 2
Imprima (MSG2);
Caso ERRO = 3
Imprima (MSG3);
Senão
Imprima (“ERRO NÃO CONHECIDO”);
Fim Conforme;
-
Fim.
```

b) Início /* trecho de algoritmo para impressão da situação de um aluno a partir da MÉDIA obtida ao fim do curso */

```
Conforme
  Caso MÉDIA > 7.0
  Imprima (“APROVADO COM DIPLOMA”);
  Caso MÉDIA > = 5.0
  Imprima (“APROVADO COM CERTIFICADO”);
  Senão
    Imprima (“REPROVADO”);
Fim Conforme;
-
Fim.
```

4. VETORES

Um vetor ou agregado homogêneo, ou ainda variável composta homogênea, é uma estrutura de dados que contêm elementos de mesmo tipo, que podem ser referenciados como um todo. Ao declararmos um vetor, estamos reservando na memória principal do computador uma série de células para uso da variável daquele tipo. O nome do vetor aponta para a base das células e o seu início dá a posição relativa do elemento referenciado ao primeiro (base).

Nem sempre os tipos básicos (inteiro, real, caractere e lógico) são suficientes para exprimir estruturas de dados em algoritmos. Por exemplo consideremos um problema em que umprofessor com 5 alunos deseja imprimir a nota e a média de seus alunos. Nesse caso serianecessário se considerar cinco variáveis reais para contar as notas dos cinco alunos. Imagine queo número de alunos da turma seja 80. Só a declaração destas variáveistornaria impraticável aredação do algoritmo. Daí a necessidade de novos tipos serem criados. Um destes tipos, o vetor,será estudado.

Os vetores podem ser unidimensionais ou multidimensionais (matrizes). Um vetorunidimensional, como uma lista de notas dos 50 alunos de uma turma, tem apenas um índice. Seexistirem porém várias turmas poderemos utilizar um vetor com dois índices (o número da turma eo número do aluno da turma). Abordaremos este assunto ainda neste capítulo.

4.1. DECLARAÇÃO DE VETORES

tipo_da_variável nome_da_variável [Pi:Pf];
Pi - Posição Inicial do Vetor
Pf - Posição Final do Vetor
Quando o C vê uma declaração como esta ele reserva um espaço na memóriasuficientemente grande para armazenar o número de células especificadas em tamanho. Porexemplo, se declararmos:

float exemplo [20];

o 'C' irá reservar 4x20=80 bytes. Estes bytes são reservados de maneira contígua. Nalinguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, osdados serão indexados de 0 a 19. Para acessá-los vamos escrever:

exemplo[0]
exemplo[1]
.
.
exemplo[19]

Mas ninguém o impede de escrever:

exemplo[30]
exemplo[103]

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos.Este é um cuidado que ***você*** deve tomar. Se o programador não tiver atenção com os limites devalidade para os índices ele corre o risco de ter variáveis sobrescritas ou de ver o computadortravar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

Exemplo:

Real Notas[1:80];

O número de elementos de um vetor é dado por: **Pf-Pi+1**
Isto significa que as posições do vetor são identificadas a partir de Pi, com incrementos unitários, até Pf:

Pi	Pi + 1	- - - - - - - - - - -	Pf

Cada elemento de um vetor é tratado como se fosse uma variável simples. Para referência a umelemento do vetor, utiliza-se o nome do vetor e a identificação do elemento (índice) entrecolchetes. Por exemplo, se desejarmos atribuir o valor "FMI" ao elemento identificado pelo índice6 do vetor anterior, teremos:

NOME[4] ←"FMI" que produzirá

			FMI	
1	2	3	4	5

Ex.: Algoritmo para ler as notas de 3 alunos e imprimir os nomes daqueles cujas notas foremmaior que a média.

• **Algoritmo Notas {Sem Vetor}**

```
Início
Caracter Nome1, Nome2, Nome3;
  Real Nota1, Nota2, Nota3;
Leia(Nome1, Nota1);
Leia (Nome2, Nota2);
Leia (Nome3, Nota3);
Média = (Nota1+Nota2+Nota3)/3.0 ;
Se Nota1 > Média Então
Imprima (Nome1);
Fim Se;
Se Nota2 > Média Então
Imprima (Nome2);
Fim Se;
Se Nota3 > Média Então
Imprima (Nome3);
```

• **Algoritmo Notas {Com Vetor}**

```
Início
Caracter Nome[0:2]
Real Nota[0:2];
  Real Soma;
Para I de 0 até 2 Faça
Leia (Nome[I], Nota[I]);
Fim Para;
/*Cálculo da Média*/
Soma = 0.0 ;
Para I de 0 até 2 Faça
    Soma = Soma + Nota [I] ;
  Fim Para;
  Média = Soma/3.0;
  Para I de 0 até 2 Faça
    Se Nota[I] > Média Então
```

```
Fim Se;                                                    Imprima (Nome[I]);
Fim.                                                     Fim Se;
                                                      Fim Para;
                                                      Fim.
```

1- Para inicializar um vetor com valores diferentes, utilizamos um comando " construtor"
NOTA ={ 5.7, 9.5, 10.0, ..., 7.5}; que equivale a: NOTA [0] = 5.7, NOTA[2] =10.0;....etc.
1) Dado o vetor CRR de caracteres abaixo,
CRR

!	U	O	T	R	E	C	A
0	1	2	3	4	5	6	7

Qual será a sua configuração depois de executados os comandos:

```
Para I de 0 até 3 passo 1 Faça
AUX =CRR[I];
CRR[I]=CRR[6-I+1];
CRR[6-I+1] =AUX;
Fim Para;
```

2) Dados dois vetores R[9] e S[19], escreva um algoritmo que: Leia os vetores R e S e Gere ovetor X correspondente à união dos vetores R e S.

```
Início
  Inteiro R[0:9],S[0:19],X[0:29];
Inteiro I;
  Leia (R, S);
Para I de 0 até 9 Passo 1 Faça
X[I] = R[I];
Fim Para
Para I de 0 até 19 Passo 1 Faça
    X[I+10] = S[I];
  Fim Para;
  Imprima (X);
Fim.
```

a) Gere o vetor Y com os elementos comuns de R e S

```
Início
  inteiro R[0:9], Y[0:9], S[0:19], I, J, K;
  leia (R,S);
Y = 0
para I de 0 até 9 faça
K = 1;
para J de 0 até 19 faça
se R[I] == S[J] então
enquanto K<=I faça
          se Y[K] != R[I] então
se Y[K]==0 então
Y[K] =R[I];
fim se;
senão abandone;
fim se;
K =K+1;
fim enquanto;
fim se;
      se Y[K]==R[I] então
abandone;
fim se;
fim para;
fim para;
imprima (Y);
Fim.
```

3) Um armazém trabalha com 100 mercadorias diferentes identificadas pelos números inteiros de 1 a 100. O dono do armazém anota a quantidade de cada mercadoria vendida durante o mês. Ele tem uma tabela que indica para cada mercadoria o preço de venda. Escreva o algoritmo para calcular o faturamento mensal de armazém, isto é

$$FATURAMENTO = \sum_{i=1}^{100} (QUANTIDADE_i * PREÇO_i)$$

A tabela de preço e quantidade são fornecidas em dois conjuntos, sendo que um conjunto contém a quantidade vendida e o outro o preço de cada mercadoria.

Solução

```
Início  /* calcula o faturamento*/
  real QUANTIDADE[0:99], PREÇO[0:99] /*Tabela de Qtdes vendidas e Preço*/
  real FATURAMENTO;
  inteiro I; /* indexador*/
  FATURAMENTO = 0;
  leia (QUANTIDADE, PREÇO);
  para I de 0 até 99 faça
    FATURAMENTO = FATURAMENTO +QUANTIDADE[I]*PREÇO[I];
  fim para;
  imprima (FATURAMENTO);
Fim.
```

3.10. EXERCÍCIOS SOBRE VETORES

1. Faça um algoritmo que leia 18 números inteiros e calcule e imprima a soma dos mesmos.

2. Faça um algoritmo que leia 18 números inteiros e calcule e imprima a média dos mesmos

3. Faça um programa que leia 4 números inteiros e apresente:
   - Média dos ímpares
   - Maior número par
   - Diferença do maior menos o menor número

4. Faca um programa que leia um conteúdo com 100 números inteiros e apresente a media dosnúmeros.

5. Faca um programa que leia 2 conjuntos com 50 números inteiros e gere um terceiro conjuntocom a media dos elementos de A e B. Apresente o C.

6. Faca um programa que leia um conjunto A com 30 números inteiros e que armazene em umconjunto B do mesmo tipo os elementos de A multiplicados por 3. apresente o conjunto B.

7. Faça um programa que leia um conjunto A com 100 números inteiros e que gere um conjunto Bcom os elemento de A de acordo com a seguinte regra. Se o elemento de A for par, armazene emB multiplicando por 5. Se o elemento de A for impar, armazene em B somando 5.

8. Faça um programa que leia dois conjuntos A e B com 20 inteiros e troque os valores de A comB.

9. Faça um programa que leia dois conjuntos A e B com 15 números inteiros e gere um conjuntode 30 elementos, onde os 15 primeiros elementos são de A e o resto de B. Apresente oconjunto C.

5. MATRIZES

Definição: Conjunto de dados referenciados por um mesmo nome e que necessita de mais deum índice para ter seus elementos individualizados.

• Quando temos mais de uma dimensão então utilizamos as matrizes, que de diferente, nouso e na declaração, só tem estas dimensões a mais explícitas:

• A sintaxe é a seguinte:
<Tipo básico> identificador [Pi1:Pf1, Pi2:Ps2,....., Pin:Pfn];

Esta especificação corresponde apenas à criação do modelo, e, para efetivar a estrutura dedados dentro do algoritmo, é necessária uma declaração dando um nome à variável que segue omodelo.

Ex:
Real MAT[1:3,1:4];
ou seja,

MAT

	1	2	3	4
1				
2				
3				

O número de dimensões da matriz será igual ao número de vírgulas (,) da declaraçãomais 1. O número de elementos é igual ao produto do número de elementos de cadadimensão: ***(Pf1-Pi1+1)*(Pf2-Pi2+1)*...*(Pfn-Pin+1)***

Exercícios

1) Qual é o número de elementos e dimensões das matrizes especificadas abaixo:

a) real MAT1[0:2,1:5];

b) Caracter MAT2[1:3, 2:4, 3:4]

Para referenciar um elemento da matriz são necessários tantos índices quantas são as dimensõesda matriz. Numa matriz bidimensional (duas dimensões) o primeiro índice indica a linha e osegundo, a coluna. No exemplo anterior, MAT1[1,3] se refere ao elemento da linha número 1(segunda linha no caso) e coluna número 3 (terceira coluna)

MAT1

	1	2	3	4	5
0					
1					
2					

Para matrizes com três dimensões, repete-se a estrutura bidimensional tantas vezes quantos sãoos elementos da terceira dimensão, numerando-as de acordo com os limites especificados nadeclaração de tipo.

Exemplo1:O que será impresso no algoritmo abaixo:

```
Início
inteiro M1[1:3,1:2];
inteiro: I, J;
M1[1,1] = 1;
M1[1,2] = 2;
M1[2,1] = 3;
M1[2,2] = 4;
M1[3,1] = 5;
M1[3,2] = 6;
Imprima(M1);
```

```
Para I de 1 até 2 Faça
Para J de 1 até 3 Faça
Imprima (M1[J,I]);
Fim Para;
Fim Para;
Fim.
```

Exemplo 2:

Dada a matriz MAT abaixo

	1	2	3	4
1	O	Q	*	I
2	E	A	E	S
3	A	*	*	S

Qual será a configuração de MAT depois de executado o algoritmo:

```
Início
Inteiro I, J;
Caracter AUX;
Caracter M1[1:4,1:4];
Leia (MAT);
Para I de 1 até 4 Passo 1 Faça
Para J de I+1 até 4 Passo 1 Faça
AUX = MAT[I,J];
MAT[I,J] = MAT[J,I];
MAT[J,I] = AUX;
Fim Para;
Fim Para;
AUX=MAT[1,1];
MAT[1,1]=MAT[4,4];
MAT[4,4]= AUX;
AUX = MAT[2,2];
MAT[2,2] = MAT[3,3];
MAT[3,3] = AUX
Fim.
```

Notas:

1- Para percorrer a matriz linha por linha:
Fixar a linha
Variar a coluna

```
Inteiro exemplo[1:3,0:4];
I =1;
Enquanto I<=3 Faça
J=0;
Enquanto J<= 4 Faça
exemplo[I,J] =I+J;
J=J+1;
Fim Enquanto;
I=I+1;
Fim Enquanto;
```

2- Para percorrer a matriz coluna por coluna
Fixar a coluna
Variar a linha

```
Inteiro exemplo[1:3,0:4];
J =0;
Enquanto J<=4 Faça
I=1;
```

```
Enquanto I<= 3 Faça
exemplo[I,J] =I+J;
I=I+1;
Fim Enquanto;
J=J+1;
Fim Enquanto;
```

ou então:

```
Para j de 0 até 4 Passo 1 Faça
Para I de 1 até 3 Passo 1 Faça
exemplo[I,J]←I+J;
Fim Para;
Fim Para;
```

Como no caso de vetores, também para matrizes podemos ter comandos concisosparainicialização, leitura e impressão.

Por exemplo, seja MAT definida por:

Inteiro MAT[1:10,1:8];

O comando:

Leia (MAT);

é equivalente ao seguinte trecho:

```
Para i de 1 até 10 Passo 1 Faça
Para j de 1 até 8 Passo 1 Faça
leia (MAT[I,J];
Fim Para;
Fim Para;
```

Do mesmo modo, pode-se inicializar uma matriz com todos os seus elementos iguais a umdeterminado valor escrevendo

MAT =0;

5.1. Exercícios Resolvidos

1) Dada uma matriz MAT de 4 x 5 elementos, faça um algoritmo para somar os elementos de cadalinha gerando o vetor SOMALINHA. Em seguida, somar os elementos do vetor SOMALINHA navariável TOTAL que deve ser impressa no final:

Exemplo

1	*2*	*-1*	*2*	*3*	*7*
1	*3*	*4*	*2*	*0*	*10*
8	*5*	*1*	*3*	*2*	*19*
1	*- 2*	*3*	*4*	*5*	*11*

TOTAL 47

Solução:

```
Início
Real MAT[1:4, 1:5];
Real SOMALINHA[1:4];
Real: TOTAL;
Inteiro: I, J;
SOMALINHA=0;
TOTAL=0;
```

```
Leia (MAT);
Para I de 1 até 4 Faça
Para J de 1 até 5 Faça
SOMALINHA[I]= SOMALINHA[I]+MAT[I,J];
Fim Para;
TOTAL=TOTAL+SOMALINHA[I];
Fim Para;
Imprima (“TOTAL=”, TOTAL);
Fim.
```

2) Escreva um algoritmo que leia duas matrizes reais de dimensão 3 x 5, calcule e imprima asoma das matrizes.

```
Início
Real A[1:3,1:5], B[1:3,1:5], C[1:3,1:5];
Inteiro: I, J;
Leia (A,B);
I=1;
Enquanto I<=3 faça
J=1
Enquanto J<=5 faça
C[I,J] =A[I,J]+B[I,J];
J=J+1;
Fim Enquanto;
I=I+1;
Fim Enquanto;
Imprima ( “C=” ,C );
Fim.
```

3) Escreva um algoritmo para um programa que leia uma matriz quadrada 20 x 20 de elementosreais, divida cada elemento de uma linha da matriz pelo elemento da diagonal principal desta linhae imprima a matriz assim modificada.

Obs. $\begin{bmatrix} \# & \\ & \# \end{bmatrix}$ Elementos da diagonal principal; M[1,1], M[2,2], M[3,3], M[4,4]

```
Início
real M[1:20,1:20];
Inteiro: I, J;
Real: DIAGONAL;
Leia (M);
Para I de 1 até 20 Faça
DIAGONAL =M[I,I];
Para J de 1 até 20 Faça
M[I,J] =M[I,J]/DIAGONAL;
Fim Para;
Fim Para;
Imprima (“M=”,M);
Fim.
```

MODULARIZAÇÃO DE ALGORITMOS

1. INTRODUÇÃO

Vimos que os algoritmos estruturados são desenvolvidos levando-se em conta algumaspremissas básicas:

1) Desenvolver o algoritmo em diferentes fases de detalhamento crescente, do geral ao particular,por refinamentos sucessivos (desenvolvimento "top-down" ou de cima para baixo).

2) Decompor o algoritmo em módulos funcionais, organizados de preferência em um sistemahierárquico. Esses módulos trazem vantagens adicionais para testes, pois testa-se um módulo (ouseja, uma parte do programa) de cada vez, independentemente; e para reutilização de um móduloem outros algoritmos e programas, no futuro.

Passaremos agora a formalizar a utilização de módulos nos nossos algoritmos e verificar asvantagensque os mesmos trazem para entendimento, construção, codificação, teste e reutilizaçãodos mesmos. A grande maioria das linguagens de programação que são utilizadas, tem estafacilidade, seja com o nome de Sub-rotinas, Subprogramas, Procedimentos, Funções, Módulos,Blocos, etc., sempre é possível subdividir-se um programa de modo a facilitar o entendimento,permitir a reutilização, evitando-se a repetição de blocos dos programas. No nosso pseudo-códigodefiniremos dois tipos de módulos: Os procedimentos ("procedures") e as funções ("functions").

Os procedimentos e funções são normalmente definidos antes de seremutilizados(chamados) pelo programa principal. Em se tratando de algoritmos, entretanto,poderemos definir nossos procedimentos e funções em qualquer parte do algoritmo principal oudepois dele, adotando os formatos que se seguem e as normas de documentação.

2. PROCEDIMENTOS

Um procedimento é uma seqüência de comandos precedida por uma seqüência dedeclarações que possui um identificador (nome do procedimento), uma lista de parâmetrosopcional e pode realizar qualquer tipo de processamento que o programador ou analista deseje.As variáveis, os tipos e as constantes declaradas dentro de um procedimento só são acessáveisdentro dos comandos do procedimento. São chamadasvariáveis locais. Em um Algoritmo, asvariáveis, tipos e constantes declaradas logo após o identificador do algoritmo, são acessíveis evisíveis dentro de qualquer procedimento que esteja sendo usado pelo algoritmo. São chamadasvariáveis Globais. É uma boa técnica ao se construir um procedimento, não fazer uso no mesmo,de variáveis globais e referenciar e utilizar apenas as variáveis locais e os parâmetros doprocedimento. Isso fará com que cada procedimento, ao ser modificado, não afete outras partesdo Algoritmo, e da mesma forma as variáveis globais poderão ser modificadas sem que hajaefeitos colaterais nos procedimentos.

Sintaxe de declaração

Procedimento < nome do Procedimento>

```
Início
<declarações>;
C1;
C2;
.
Cn;
Fim { nome do procedimento}.
```

Exemplo: O que será impresso no algoritmo abaixo?

```
Início
Inteiro: X,Y,A,B,C,D;
Procedimento troca;
Início
Inteiro: AUX,X;
AUX←X;
X←Y;
Y←AUX;
Fim troca.
A←5;
B←3;
```

```
Imprima(A, B);
X←A;
Y←B;
Troca;
A←X;
B←Y;
Imprima(A, B);
C←4;
D←9;
Imprima(C, D);
X←C;
Y←D;
Troca;
C←X;
D←Y;
Imprima(C, D);
Fim.
```

Os módulos que estivemos utilizando na apostila, nos capítulos anteriores, são procedimentossem parâmetros, ou seja, que fazem uso de variáveis globais e sua única importante vantagem éfacilitar o entendimento e a solução do algoritmo.

Utilizando parâmetros e não variáveis globais dentro do procedimento, podemos, por assimdizer, isolá-lo do meio exterior e obter vantagens adicionais na codificação, pois, uma vez queforam bem definidos os parâmetros, um programador poderá desenvolver seu procedimento semse preocupar com o algoritmo que o vai chamar. Poderemos também testá-lo individualmente eisoladamente para verificar sua correção. Isto será comprovado na disciplina LP. A isto sedenomina Encapsulamento. Uma lista de parâmetros consiste em uma lista de variáveis e seusrespectivos tipos.

```
Procedimento < nome do Procedimento> (<lista de parâmetros>)
< especificação de parâmetros>
Início
<declaração de variáveis locais>
C1;
C2;
.
.
Cn;
Fim. { nome do procedimento}
```

Exemplo de Procedimento:

```
Procedimento TROCAINTEIROS (Inteiro NUM1,inteiro NUM2)
Início
Inteiro AUXNUM;
AUXNUM = NUM1;
NUM1 = NUM2;
NUM2 = AUXNUM;
Fim.
```

3. FUNÇÕES

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Senão as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Parafazermos programas grandes e complexos temos de construí-los bloco a bloco.

Uma função é um módulo que tem por objetivo calcular e retornar ao algoritmo,procedimento ou função "chamador" um valor. A chamada de uma função para execução é feitaapenas citando-se o seu identificador no meio de uma expressão. Vide as funções previamenteexistentes no nosso pseudo-código. Uma função é dita recursiva quando chama a ela própria.

Uma função no C tem a seguinte forma geral:
tipo_de_retorno nome_da_função (declaração_de_parâmetros)
Início

corpo_da_função
Fim.

Em PORTUGOL utilizaremos a seguinte forma

• **PORTUGOL**
Tipo-de-retorno<nome da função>(<declaração dos parâmetros>)
Início
Corpo da Função
Fim.

Onde:
• A lista de parâmetros é semelhante à dos procedimentos
• Antes do nome da função dá o tipo do valor que a função retorna ao algoritmo ouprocedimento chamador.

O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo **inteiro(int)**, ou seja, uma função para qual não declaramos o tipo de retorno é considerada comoretornando um inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:

tipo nome1, tipo nome2, ... , tipo nomeN

Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É nadeclaração de parâmetros que informamos ao compilador quais serão as entradas da função(assim como informamos a saída no tipo-de-retorno).
O corpo da função é a sua alma. É nele que as entradas são processadas, saídas sãogeradas ou outras coisas são feitas.
O Comando retorna (return)
O comando **return** tem a seguinte forma geral:

retorna valor_de_retorno; ou *retorna;*

Digamos que uma função está sendo executada. Quando se chega a uma declaração **retorna (return)** a função é encerrada imediatamente e, se o valor de retorno é informado, afunção retorna este valor. É importante lembrar que o valor de retorno fornecido tem que sercompatível com o tipo de retorno declarado para a função.

Uma função pode ter mais de uma declaração **retorna (return)**. Isto se torna claro quandopensamos que a função é terminada quando o programa chega à primeira declaração retorna **(return)**. Abaixo estão dois exemplos de uso do retorna (**return)**:

Exemplo de função:

```
Inteiro VALORABSOLUTO (inteiro X)
Início
Se X > = 0 Então
VALORABSOLUTO =X;
Senão
VALORABSOLUTO = (-X);
Fim Se;
Fim.
```

Exercício para casa: Escrever e executar em C o programinha acima

- Algoritmo para calcular o quadrado de um número

• **PORTUGOL**

```
Inteiro Square(inteiro a)
Início
retorna (a*a);
Fim.

Algoritmo Principal

Início
inteiro num;
imprima ("Entre com um numero: ");
```

```
leia (num);
num=Square(num);
imprima ("O seu quadrado vale:", num);
Fim.
```

• **Em C**

```
#include <stdio.h>
int Square (int a){
return (a*a);
}
int main (){
int num;
printf ("Entre com um numero: ");
scanf ("%d",&num);
num=Square(num);
printf ("\n\nO seu quadrado vale: %d\n",num);
return 0;
}
```

- Algoritmo para verificar se um número **a** é divisível por 2.

• **PORTUGOL**

```
inteiro EPar (inteiro a)
início
se (a%2) então /* Verifica se a e divisivel por dois */
return 0; /* Retorna 0 se nao for divisível, a%1=1 */
senão
return 1; /* Retorna 1 se for divisivel , a%0=0*/
fim se;
fim.

Algoritmo Principal

Início
inteiro num;
imprima ("Entre com numero: ");
leia (num);
se (EPar(num)) então
imprima ("O numero e par.");
senão
imprima ("O numero e impar.");
fim se;
Fim.
```

• **Em C**

```
#include <stdio.h>
int EPar (int a){
if (a%2) /* Verifica se a e divisivel por dois */
return 0; /* Retorna 0 se nao for divisivel */
else
return 1; /* Retorna 1 se for divisivel */
}
int main (){
int num;
printf ("Entre com numero: ");
scanf ("%d",&num);
if (EPar(num))
printf ("\n\nO numero e par.\n");
else
printf ("\n\nO numero e impar.\n");
return 0;
}
```

É importante notar que, como as funções retornam valores, podemos aproveitá-los parafazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemosfazer:

```
func(a,b)=x; /* Errado! */
```

4. Protótipos de Funções

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função principal, **main()**. Isto é, as funções estão fisicamente antes da função principal **main()**.

Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto que as funções foram colocadas antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos. Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência? A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:

• **PORTUGOL**

```
inteiro Square (inteiro a);

Algoritmo Principal

início
inteiro num;
imprima ("Entre com um numero: ");
leia (num);
num=Square(num);
imprima ("O seu quadrado vale:", num);
retorna (0);
fim.

inteiro Square (inteiro a)

início
retorna (a*a);
fim.
```

• **'C'**

```
#include <stdio.h>
int Square (int a);
int main (){
int num;
printf ("Entre com um numero: ");
scanf ("%f",&num);
num=Square(num);
printf ("\n\nO seu quadrado vale: %f\n",num);
return 0;
}

int Square (int a){
return (a*a);
}
```

Observe que a função **Square()** está colocada depois do algoritmo principal, mas o seuprotótipo está antes. Sem isto este programa não funcionaria corretamente.

Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveisÉ bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Osprotótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, ocompilador evita erros, não deixando que o programador use funções com os parâmetros erradose com o tipo de retorno errado, o que é uma grande ajuda!

5. Escopo de Variáveis

O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nasdiversas partes do programa.

5.1. Variáveis locais

O primeiro tipo de variáveis que veremos são as variáveis locais. Estas são aquelas que sótêm validade dentro do bloco no qual são declaradas. Podemos declarar variáveis dentro dequalquer bloco. Só para lembrar: um bloco começa quando abrimos uma chave e termina quandofechamos a chave. Até agorasó tínhamos visto variáveis locais para funções completas. Mas umcomando para **(for)** pode ter variáveis locais e que não serão conhecidas fora dali. A declaraçãode variáveis locais é a primeira coisa que devemos colocar num bloco. A característica que tornaas variáveis locais tão importantes é justamente a de serem exclusivas do bloco. Podemos terquantos blocos quisermos com uma variável local chamada **x**, por exemplo, e elas nãoapresentarão conflito entre elas.

A palavra reservada do C **auto** serve para dizer que uma variável é local. Mas nãoprecisaremos usá-la pois as variáveis declaradas dentro de um bloco já são consideradas locais.

Abaixo vemos um exemplo de variáveis locais:

• **PORTUGOL**

```
funcao1 (...)
Início
inteiro abc,x;
...
Fim.

funcaq (...)
Início
inteiro abc;
...
Fim.

inteiro principal ()
Início
inteiro a,x,y;
para (...)
{
real a,b,c;
...
}
...
Fim.
```

• **‘C’**

```
func1 (...)
{
int abc,x;
...
}

func (...)
{
int abc;
```

```
...
}

int main ()
{
int a,x,y;
for (...)
{
float a,b,c;
...
}
...
}
```

No programa acima temos três funções. As variáveis locais de cada uma delas não irãointerferir com as variáveis locais de outras funções. Assim, a variável **abc** de **func1()** não temnada a ver (e pode ser tratada independentemente) com a variável **abc** de **func2()**. A variável **x** de**func1()** é também completamente independente da variável **x** da função principal **main()**. Asvariáveis **a**, **b** e **c** são locais ao bloco para**(for)**. Isto quer dizer que só são conhecidas dentrodeste bloco e são desconhecidas no resto da função Principa**(main())**. Quando usarmos avariável **a** dentro do bloco para **(for)** estaremos usando a variável **a** local ao para**(for)** e não avariável **a** da função **main()**.

3.11. Parâmetros formais

O segundo tipo de variável que veremos são os parâmetros formais. Estes são declaradoscomo sendo as entradas de uma função. Não há motivo para se preocupar com o escopo deles. Éfácil: o parâmetro formal é uma variável local da função. Você pode também alterar o valor de umparâmetro formal, pois esta alteração não terá efeito na variável que foi passada à função. Istotem sentido, pois quando o C passa parâmetros para uma função, são passadas apenas cópiasdas variáveis. Isto é, os parâmetros formais existem independentemente das variáveis que forampassadas para a função. Eles tomam apenas uma cópia dos valores passados para a função.

3.12. Variáveis globais

Variáveis globais são declaradas, como já sabemos, fora de todas as funções do programa.Elas são conhecidas e podem ser alteradas por todas as funções do programa. Quando umafunção tem uma variável local com o mesmo nome de uma variável global a função daráprefer̂ncia à variável local. Vamos ver um exemplo no nosso pseudo-Código:

```
inteiro z,k;
func1 (...)
{
inteiro x,y;
...
}
func2 (...)
{
inteiro x,y,z;
...
z=10;
...
}
principal
{
inteiro count;
...
}
```

No exemplo acima as variáveis **z** e **k** são globais. Veja que **func2()** tem uma variável localchamada **z**. Quando temos então, em **func2()**, o comando **z=10** quem recebe o valor de 10 é avariável *local*, não afetando o valor da variável global **z**.Evite *ao máximo* o uso de variáveis globais. Elas ocupam memória o tempo todo (as locaissó ocupam memória enquanto estão sendo usadas) e tornam o programa mais difícil de serentendido e menos geral.

6. Passagem de parâmetros por valor e passagem por referência

Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais dafunção copiam os valores dos parâmetros que são passados para a função. Isto quer dizer quenão são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada defunção é denominado chamada por valor. Isto ocorre porque são passados para a função apenasos valores dos parâmetros e não os próprios parâmetros. Veja o exemplo abaixo:

Exemplo

• **PORTUGOL**

```
real sqr (real num);
int main ()
{
real num,sq;
imprima ("Entre com um numero: ");
leia (num);
sq=sqr(num);
imprima ("O numero original e:",num);
imprima ("O seu quadrado vale:",sq);
}

real sqr (real num)
{
num=num*num;
retorna num;
}
```

• **Em C**

```
#include <stdio.h>
float sqr (float num);
void main ()
{
float num,sq;
printf ("Entre com um numero: ");
scanf ("%f",&num);
sq=sqr(num);
printf ("\n\nO numero original e: %f\n",num);
printf ("O seu quadrado vale: %f\n",sq);
}

float sqr (float num)
{
num=num*num;
return num;
}
```

No exemplo acima o parâmetro formal **num** da função **sqr()** sofre alterações dentro dafunção, mas a variável **num** da função principal **main()** permanece inalterada: é uma chamada porvalor.

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nosparâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passadospara a função. Este tipo de chamada de função tem o nome de "chamada por referência". Estenome vem do fato de que, neste tipo de chamada, não se passa para a função os valores dasvariáveis, mas sim suas referências (a função usa as referências para alterar os valores dasvariáveis fora da função).

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais àvontade dentro da função, sem termos que nos preocupar em estar alterando os valores dosparâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porquepodemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recursoque permite ao programador fazer chamadas por referência. Há entretanto, no C, um recurso deprogramação que podemos usar para simular uma chamada por referência. Quando queremosalterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetrosformais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterara variável fora da função. O

único inconveniente é que, quando usarmos a função, teremos delembrar de colocar um**&**na frente das variáveis que estivermos passando para a função. Veja umexemplo:

```
#include <stdio.h>
void Swap (int *a,int *b);
void main (void)
{
int num1,num2;
num1=100;
num2=200;
Swap (&num1,&num2);
printf ("\n\nEles agora valem %d %d\n",num1,num2);
}

void Swap (int *a,int *b)
{
int temp;
temp=*a;
*a=*b;
*b=temp;
}
```

Não é muito difícil. O que está acontecendo é que passamos para a função Swap oendereço das variáveis num1 e num2. Estes endereços são copiados nos ponteiros a e b. Atravésdo operador * estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Mas,quem é este conteúdo? Nada mais que os valores armazenados em num1 e num2, que, portanto,estão sendo modificados!

7. FUNÇÕES RECURSIVAS

Existem casos em que um procedimento ou função chama a si próprio. Diz-se então que oprocedimento ou função é recursivo. Por exemplo, o fatorial de um número n pode ser definidorecursivamente, ou seja:

$$n! = \begin{cases} n * (n-1)! \\ 1 \; se \; n = 0 \end{cases}$$

```
inteiro fat(inteiro n)
Início
Se n==0 Então
fat=1;
Senão
fat=n*fat(n-1);
Fim Se;
Fim.
```

Por exemplo, o fatorial de 3 será calculado a partir da chamada à função pelo comando x =fat(3) que retornará em fat o valor 6.

8. ESTRUTURAÇÃO DOS MÓDULOS DE UM ALGORITMO

Um procedimento nada mais é que um algoritmo hierarquicamente subordinado a um outroalgoritmo, comumente chamado de Módulo principal ou programa principal ou ainda algoritmoprincipal.

Da mesma forma, um procedimento poderá conter outros procedimentos (e também funções)aninhados. Assim, ao definirmos as fases do nosso algoritmo, no processo de refinamentossucessivos, se transformarmos as mesmas em procedimentos e os refinamentos decada procedimentoem outros procedimentos, obteremos, ao fim do processo de criação, umalgoritmo estruturado, com módulos hierarquicamente subordinados e muito menos suscetível aerros de lógica na fase de codificação do programa.

ALGORITMOS DE PESQUISA E ORDENAÇÃO

1. INTRODUÇÃO

Quando temos um Vetor (ou Matriz) com muitos elementos e precisamos descobrir se umdeterminado elemento que procuramos se encontra no vetor, uma solução que certamente nosvem à mente é comparar o elemento que procuramos com cada elemento do vetor, até queencontremos ou até que concluamos que o elemento procurado não está no vetor.

Esta é a base do raciocínio dos algoritmos de pesquisa ou busca ("Search"), que comosugere o nome, "Pesquisam", em um vetor, a existência ou não existência de um elementoprocurado. A diferença entre um e outro algoritmo de busca, fica por conta da rapidez com que"varremos" o vetor para encontrar o elemento ou para concluirmos que ele não existe.

Um fator que influencia em muito nessa rapidez é a disposição dos elementos no vetor. Seestão desordenados, seguramente teremos que verificar do primeiro ao último elemento paraconcluir, com certeza, que o elemento não existe. Já se estão ordenados, ao encontrarmos umelemento maior (ou menor) que o elemento procurado, poderemos concluir pela sua nãoexistência. Os algoritmos de ordenação ou classificação ("Sort"), por sua vez, são utilizados paraordenar os elementos de um vetor de forma a facilitar a pesquisa posterior de um elemento, noconjunto de elementos existentes.

Existem algoritmos para pesquisa e ordenação para as mais variadas estruturas de dados. Nanossa disciplina, entretanto, trataremos apenas os algoritmos de pesquisa e ordenação emvetores, que também podem ser utilizados em matrizes, desde que sofram pequenos ajustes.

2. ALGORITMOS DE PESQUISA

Para fazermos qualquer pesquisa em vetor (ou matriz) precisamos de quatro parâmetros:

a) O vetor no qual realizaremos a pesquisa

b) O número de elementos desse vetor que devem ser pesquisados. (Lembre-se que muitasvezes um vetor de 1000 elementos, só tem 700 carregados, e podemos evitar tratamento de 300elementos com "lixo").

c) O elemento procurado

d) Um índice que vai ser preenchido com a posição onde o elemento foi encontrado ou retornarácom 0 (zero) caso o elemento não exista.

Como os algoritmos de pesquisa poderão ser utilizados muitas vezes na solução de diferentesproblemas, vamos defini-los como Procedimentos de um Algoritmo Principal hipotético, com osseguintes argumentos:

```
Declarações:
Const
MAXELEM = 10000 {Número máximo de elementos do vetor, apenas para limitar o tamanho dovetor}
<Tipo básico> Nome do vetor[1:MAXELEM]
inteiro TOTELEM {Corresponde ao parâmetro b) Total de elementos a ser pesquisado}
<Tipo básico> ELEMPROC {Corresponde ao Parâmetro c) Elemento procurado}
Inteiro POS {Corresponde ao Parâmetro d) Posição em VET onde ELEMPROC foi encontrado, ou0 se não o foi}
```

2.1. PESQUISA SEQÜENCIAL SIMPLES

Na pesquisa seqüencial simples, como o vetor a ser pesquisado não está ordenado peloelemento procurado, teremos de comparar um a um o ELEMPROC com cada elemento de VET.Portanto para um elemento inexistente teremos de fazer TOTELEM testes e para um elementoexistente faremos, na média, TOTELEM/2 testes.

```
Procedimento PESQSEQ (<tipo básico>VET);
    Início
    Inteiro TOTELEM, POS, J;
    <Tipo básico> ELEMPROC;
```

```
Leia(ELEMPROC);
/*Executa a Pesquisa da primeira ocorrência de ELEMPROC em VET eretorna em POS o índice onde
foi encontrada ou 0 se não existe*/
Enquanto (Pos==0) e (J<=TOTELEM) Faça
Se VET[J]==ELEMPROC Então
POS = J;
Senão
J = J+1;
Fim Se;
Fim Enquanto;
Fim.
```

Se precisamos determinar todas as ocorrências de um elemento em um vetor, o problema sesimplifica, pois teremos que, obrigatoriamente, varrer o vetor até o fim (Elimina-se do laço o testePOS = =0), mas teremos de guardar em um vetor todas as posições onde o elemento foiencontrado ou, o que é mais usual, processaremos a ocorrência dentro do procedimento (apósPOS = J). O trecho de procedimento abaixo exemplifica:

• **Pesquisa Sequencial com Repetição**

Trata todas as ocorrências de um elemento procurado no vetor

```
Início
  POS = 0; /*Indicará ao algoritmo chamador a não ocorrência*/
  J = 1;
  Enquanto J < = TOTELEM Faça
  Se VET[J] == ELEMPROC Então
  POS = J;
/*Imprime, Acumula, etc.*/
Fim Se;
  J = J + 1;
  Fim Enquanto;
Fim.
```

3.13. PESQUISA SEQÜENCIAL ORDENADA

Para se utilizar a Pesquisa Seqüencial Ordenada, o vetor de busca tem de estar ordenadopelo campo chave da pesquisa. Com isso, ao se encontrar no vetor, um elemento maior do que oelemento procurado, poderemos abandonar a busca pois, com certeza, não mais œncontraremos. Devido a isso o número de testes para elementos existentes ou inexistentes será,na média, de TOTELEM/2, e, por isso, melhor que o anterior para o caso de elemento inexistente.

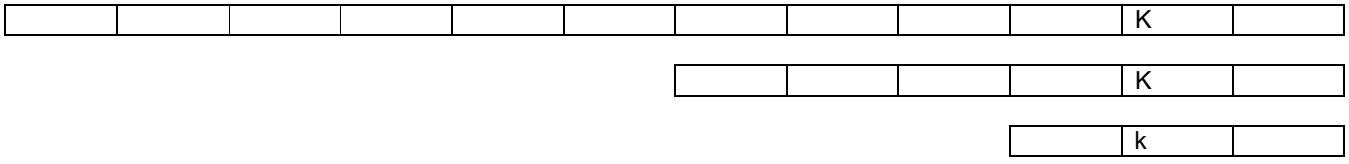
Procedimento PESQORD (<tipo básico> VET, Inteiro TOTELEM, inteiro POS, <Tipo básico>ELEMPROC);

```
Início
  Inteiro J;
  /*Executa a Pesquisa da primeira ocorrência de ELEMPROC em VET eretorna em POS o índice onde
foi encontrada ou 0 se não existe. VETtem de estar ordenado pelo campo chave*/
  POS = 0;
  J = 1;
  Enquanto J<=TOTELEM e POS==0 Faça
  Se VET[J]>=ELEMPROC Então
  Se VET[J] == ELEMPROC Então
  POS=J;
Senão
  Imprima(" O elemento não está na lista");
  Abandone;
  Fim Se;
  Senão
  J=J+1;
  Fim Se;
Fim Enquanto;
Fim.
```

Exercício resolvido - Dado um vetor A de 128 elementos, verificar se existe um elemento igual a K(chave) no vetor. Se existir, imprimir a posição onde foi encontrada a chave; se não, imprimir “chave K não encontrada” . O vetor A e chave K são lidos de cartões.

```
início
  inteiro A[1:128] inteiro;
  inteiro I,K;
  lógico ACHOU;
  leia (K);
  leia (A);
  ACHOU = falso;
  para I de 1 até 128 passo 1 faça
  se A[I]==K então
  imprima (K, “ Está na posição” , I);
ACHOU = verdadeiro;
  abandone;
  fim se;
fim para ;
  se não ACHOU então
  imprima (“A CHAVE”, K,“NÃO ESTÁ NO VETOR”);
  fim se;
fim.
```

Neste exemplo, a pesquisa a um elemento ficará mais eficiente se utilizarmos a pesquisa binária,desde que o vetor já esteja ordenado. Nesta pesquisa procuramos o elemento K dividindo o vetorem duas partes e testando em qual das duas ele deveria estar. Procedendo da mesma forma paraa parte provável, e assim sucessivamente:



3.14. PESQUISA BINÁRIA

A pesquisa binária é um método que também só se aplica a vetores previamente ordenados.A sua grande vantagem é a rapidez e, por isso, ela é muito recomendada para vetores grandes.Neste método, a primeira comparação é feita com o elemento do meio do vetor, eliminando-seassim metade do mesmo para a busca. Seguem-se comparações sucessivas ao elemento domeio do segmento onde pode estar o elemento procurado. Assim, a cada comparação feita,metade do vetor é eliminada.

```
Procedimento PESQBIN (V: VET, Inteiro: TOTELEM, POS;<Tipo básico> :ELEMPROC)
    Início
    /*Executa a Pesquisa binária da primeira ocorrência da ELEMPROC em VETe retorna em POS se o
índice foi encontrado ou 0 se não existe. VETtem de estar ordenado pelo campo chave*/
      Inteiro: PRI, ULT, MED {primeiro, último e elemento do meio}
      POS = 0;
      PRI = 1;
      ULT = TOTELEM;
      Enquanto (PRI < = ULT) e (POS == 0) Faça
      MED = (PRI + ULT)/2 /*quociente da divisão inteira*/
      Se VET[MED] == ELEMPROC Então
      POS = MED;
      Senão Se VET[MED] > ELEMPROC Então
      /*O ELEMPROC está na primeira metade*/
      ULT = MED - 1;
    Senão
      /*O ELEMPROC está na Segunda metade*/
      PRI = MED + 1;
    Fim Se;
    Fim Se;
    Fim Enquanto;
    Fim.
```

• Exercício 1 do item 7.2.2 utilizando-se pesquisa binária.

```
início /*pesquisa binária*/
  inteiro COMEÇO, /*indicador do primeiro elemento da parte do vetor aconsiderar*/
  FIM, /*indicador do último elemento da parte do vetor aconsiderar*/
  MEIO, /*indicador do elemento do meio da parte do vetorConsiderada*/
  K; /*elemento procurado*/
  Inteiro A[1:128];
  leia (A,K);
  COMEÇO =1;
  FIM=128;
  repita
  MEIO= (COMEÇO+FIM)/2;
  se K<A[MEIO] então
  FIM=MEIO-1
  senão
  COMEÇO=MEIO+1;
  fim se
  até A[MEIO]==K ou COMEÇO >FIM;
  se A[MEIO]!=K então
  imprima (“ Não existe o elemento”);
  senão
  imprima (“ Está na posição:” , MEIO);
  fim se
fim.
```

3. ALGORITMOS DE ORDENAÇÃO

Como já foi dito, o propósito dos algoritmos de ordenação é o de facilitar e acelerar abusca posterior de um elemento no vetor. Os algoritmos de ordenação são utilizadosnormalmente uma vez em cada execução do programa, ou poucas vezes, se comparadoscom os de Pesquisa, por isso o uso de métodos elementares e demorados não é tãoproblemático como nas pesquisas. Na nossa disciplina, veremos três algoritmos paraclassificação interna de vetores, que têm tempo de execução proporcional ao quadrado donúmero de elementos a serem ordenados:

- Ordenação por seleção (método seleção direta)
- Ordenação por inserção (método inserção direta)
- Ordenação por troca (método da bolha)

Para fazermos qualquer ordenação em vetor (ou matriz) precisaremos de dois parâmetros:
a) O vetor que será ordenado;
b) O número de elementos desse vetor que devem ser ordenados. (Novamente para evitartratamento de "lixo")

Da mesma forma que os algoritmos de pesquisa, os algoritmos de ordenação poderão serutilizados muitas vezes na solução de diferentes problemas e também vamos defini-los comoum Procedimento do mesmo Algoritmo Principal, hipotético, com os seguintes argumentos:

```
Declarações
Constante MAXELEM = 1000; /*Número máximo de elementos do vetor, apenas limitaro tamanho do vetor*/
<tipo básico> VET[1: MAXELEM];/*Corresponde ao Parâmetro a) o vetor a serordenado*/
Inteiro TOTLEM /*Corresponde ao Parâmetro b)*/
```

3.1. MÉTODO DE SELEÇÃO DIRETA

Este é um dos mais simples métodos existentes. Primeiro descobre-se o menorelemento do vetor e troca-se com o primeiro elemento. A partir daí, com os elementosremanescentes, repete-se o processo até que todo o vetor esteja ordenado. Um exemplobastante elucidativo é o de algumas cartas de baralho na mesa viradas para cima (à vista).Seleciona-se a menor (ou maior) para colocá-la na primeira posição. Depois, seleciona-se,sucessivamente, a menor carta dentre as remanescentes e se a coloca sobre a primeira.

```
Procedimento SELEÇÃO_DIRETA (V: VET, Inteiro: TOTELEM)

  Início
  /*Faz a ordenação crescente de TOTELEM elementos de um vetor VET*/
```

```
Inteiro MIN, I, J; /*índices*/
  <Tipo Básico> AUX;
  Para K de 1 até (TOTELEM - 1) Faça
  MIN ←K;
  Para J de (K + 1) até TOTELEM Faça
  Se VET[J] < VET [MIN] Então
  MIN = J;
  Fim Se;
  Fim Para;
  AUX = VET[MIN];
  VET[MIN] = VET[K];
  VET[K] = AUX;
  Fim Para;
Fim.
```

3.2. MÉTODO DE INSERÇÃO DIRETA

Nesse método, considera-se que um elemento do vetor (o primeiro) está ordenado E"insere-se" os demais considerando os anteriores já ordenados. Voltando-se ao exemplo dobaralho é como se agora as cartas sobre a mesa estão com a face para baixo e cada carta éarrumada ordenadamente na mão em relação as que já estão na mão.Este método pode ser utilizado, com vantagem, quando se está lendo elementos deum arquivo para serem posteriormente ordenados. Bastam leves modificações no algoritmo,que leremos o arquivo e já o armazenaremos na memória ordenadamente.

```
Procedimento INSERÇÃO_DIRETA (V: VET, Inteiro : TOTELEM)
    Início
      /*Faz a ordenação de TOTELEM elementos de um vetor VET*/
      <Tipo básico> AUX;
      Inteiro K, J; /*Índice*/
      Para K de 2 até TOTELEM Faça
      AUX = VET[K];
      J = K - 1;
      Enquanto (J > 0) Faça
      Se AUX < VET[J] Então
      VET [J + 1] = VET [J];
    J = J - 1;
    Senão
      Abandone;
      Fim Se;
      Fim Enquanto;
      VET [J + 1] = AUX;
    Fim Para;
    Fim.
```

3.3. MÉTODO DA BOLHA

Este método caracteriza-se por efetuar a comparação sucessiva de paressubseqüentes de elementos, trocando-os de posição, se estiverem fora de ordem. Dos trêseste é o menos eficiente,já que cada troca de elementos exige três comandos e o número detrocas é (TOTELEM-1) vezes maior que no método de seleção direta, por exemplo. Alémdisso o método é mais complicado de entender que o de seleção direta. Segue-se exemplode uma implementação de algoritmo de pesquisa utilizando-se esse método.

```
Procedimento BOLHACRESCENTE (V: VET, Inteiro: TOTELEM)
    Início
      /*Faz o ordenamento de TOTELEM elementos de um valor VET*/
      Inteiro P,J; /*contador de passadas e índice*/
      <Tipo básico> AUX;
      Para P de 1 até (TOTELEM - 1) Faça
      Para J de 1 até (TOTELEM - P) Faça
      Se VET[J + 1]<VET[J] Então
      AUX = VET[J]; /*Troca dos elementos*/
      VET[J] =VET[J + 1];
      VET[J + 1] =AUX;
      Fim Se;
```

```
  Fim Para;
  Fim Para;
Fim.
```

- Exercício Resolvido

1) Classificar um vetor numérico VET de 6 elementos em ordem crescente:

```
início /*método da bolha*/
inteiro VET[1:6]
inteiro: AUX, /* auxiliar para troca de elementos*/
BOLHA, /* indicador de mais alto elemento fora de ordem*/
LSUP, /*indicador do tamanho do vetor a ser pesquisado, sendo o
valor inicial igual a 6*/
J; /*indicador do elemento do vetor*/
leia (VET);
LSUP = 6;
enquanto LSUP>1 faça
BOLHA = 0;
para J de 1 até LSUP-1 faça
se VET[J]>VET[J+1] então /* troca elemento j com j+1*/
AUX = VET[J];
VET[J] = VET[J+1];
VET[J+1] = AUX;
BOLHA = J;
fim se;
fim para
LSUP = BOLHA; /*aponta para última posição trocada*/
fim enquanto;
imprima (VET);
fim.
```